User Level Page Faults

Qingyang Li

CMU-CS-20-124
December 2020

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Seth Copen Goldstein, Chair
David A. Eckhardt

Submitted in partial fulfillment of the requirements
for the degree of Masters in Computer Science.

Keywords: User level interrupt, TLB, page table, MMU, page fault

For my grandparents:
I am resting in the shade of trees that you planted.

Abstract

Memory management has always been a responsibility shared by the hardware and the operating system. The MMU (memory management unit) walks the kernel-managed page tables to ensure safety and isolation between data from different processes. While users can alter some aspects of the memory system they control (e.g., permissions) these operations come with a high overhead.

We propose User Level Page Tables as a hardware/software mechanism that offers a low overhead mechanism for users to manage their application memory permissions while still having hardware enforcing those permissions. Our mechanism allows users to modify page permissions with a single write in user space and without changing privilege levels. Users can also handle many types of page faults using handler functions that they install, without crossing into kernel space.

To realize this mechanism, we start with an architecture that has already been modified to support User-Level Interrupts and we modify components in its hardware address translation pipeline, specifically the MMU and TLB. We also modify the Linux kernel by adding a set of "shadow" user-level page tables.

We evaluate our approach of User Level Page Tables by using it to implement watchpoints and analyze the overhead. We evaluate our system using Gem5, a full system emulator, modified to implement User-Level Interrupts and User-Level Page Faults as an extension of the x86_64 architecture.

Our microbenchmarks show that, even with thousands of watchpoints, we incur a slowdown of only 3.25x compared to a 1,000,000x slowdown when using GDB software watchpoints. With the SPEC benchmarks, watching every dynamically allocated memory region, we see slowdowns range from 2.1x-15.8x. The latter slowdown suggests we need to improve our method of tracking active watchpoints.

Acknowledgments

Prof. Goldstein - Thank you for advising me these past years. I've evolved so much as a systems person under your guidance. I still have a long way to go but I now have the confidence to venture on my own. Thank you for your patience in allowing me to make mistakes. Thank you for pushing me when I was hesitant. Thank you for holding me accountable. Thank you for working on me.

Prof. Eckhardt - Thank you for providing me with resources, physical and emotional, even when I did not know I needed them. Thank you for helping me conquer my self-doubt and raise my self-esteem. I am going to miss our chats in your office.

Clem Cole, Intel Corp. - Thank you for adopting me that summer and convincing me that systems is a fun beast to play with.

Chrisma Pahka - Your patience with me is none other than god-tier. Thank you for humoring my design musings, debugging my Gem5 and compiler issues, and grounding my conceptual questions. Good luck and enjoy the rest of your PhD.

Jinghang Wang - Thank you for shouldering the hardware end of this project during the initial stages, when I barely knew what was happening. Thank you for stepping up to debug when I asked for your help.

Thank you to my roommates Caroline and Claire, for always cheering me up with some memes after a rough debugging session.

Thank you to Olivia, Alex, and Neil, for when I needed to gripe about general frustrations.

Thank you to Katie, Liam, and Doria, for getting me to play games and distracting me when I was in a rut.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is forever a tradeoff between the operating system's responsibilities and user capabilities. To enforce safety and fairness between multiple running processes, the operating system must impose some restrictions on the processes while giving the users maximal freedom. A prime example of this divide is the memory management system.

1.1 Memory Management

The operating system manages resources for multiple processes running on the same machine. Part of its responsibility is maintaining isolation between process memories; no process should be able to manipulate memory that is not its own. From the perspective of each individual process, it should seem as if it is the only process running on the machine, thus it should have access to all resources, including the entire memory address space. This conflict is resolved by using virtual addresses.

With virtual addressing, users cannot access physical memory directly but are still capable of allocating as much memory as they desire. The operating system maintains a mapping within kernel space between user-accessible virtual addresses and hardware-understandable physical addresses in the page table. This data structure is managed exclusively by the operating system; only certain capabilities are exposed to the user through system calls. Furthermore, this mapping is enforced by specialized memory translation hardware, so a write to a stray pointer or one that violates set permissions will cause the kernel to crash the application. Virtual addresses solve both the user's and system's needs; user applications have the illusion of being able to access the entire memory space through virtual addresses while being unable to manipulate another process' memory.

Using virtual addresses in user programs creates a new problem as hardware uses physical addresses to access memory. Thus, the virtual address must be translated to a physical address. Memory is allocated in contiguous chunks of 4096 bytes, known as pages. This memory corresponds to a virtual page number, when accessed by virtual addresses, and a physical page frame number, when accessed by physical addresses. The following steps take place when translating from a virtual address to a physical address.

1. The memory management unit (MMU) within the CPU will separate the virtual memory

address into two sections: the virtual page number (the first 36 bits of the address) and the offset (the remaining 12 bits of the address). The virtual page number be used in the following steps to obtain the physical page frame number. The offset will be appended to the physical page frame number to become the physical address that corresponds to the original virtual address.

2. The MMU checks the translation look-aside buffer (more commonly known as the TLB) to see whether the address has been recently translated.
   (a) If its entry is in the TLB, the physical page frame number would have been stored in the TLB.
   (b) If it is not within the TLB, the MMU will use the virtual address to index into the page tables to find the corresponding virtual address to the physical address. The resulting physical frame number that the end of the page walk will be stored in the TLB and used to obtain the physical address.

Address translation may result in several outcomes. The first, and best, case is that the physical page frame number is within the last level of the page table, and the page is mapped in memory. The MMU will return the entry's physical address, acquire the data, and execution will proceed. All other cases, signifying that either an error occurred or something needs additional attention, will throw a page fault exception and jump to the handler in kernel space.

1.2 Page Fault Handling

Under certain circumstances, the fault handler is able to recover and execution can proceed. For example, the CPU may trigger a page fault when it encounters an invalid page during address translation. Page table entries have a valid-invalid bit to indicate the page state in the context of the executing process. If the bit is marked valid, then the page is within the process' address space (i.e. the access is legal) and the page is in memory. In any other case, the page table entry for this page is marked invalid. So, a page that is within the address space of the process but is not in memory will also throw a page fault. This can occur when RAM cannot hold all memory pages for all running processes and stores the excess on some secondary storage drive [9, Chapter 9]. This can also occur when the process being run for the first time, therefore none of its data is in memory. The page fault handler in the kernel determines that the page is in storage and swaps it into memory. Once the data is in memory, the handler updates the page table to mark the page as valid. The instruction will again be executed. Since the access is now on a valid page, program execution will continue past the point of the previous fault. This is an example of a recoverable fault.

In other cases, the page fault handler is unable to resolve the cause of the exception. If the access is on kernel memory or memory outside the process' address space, the kernel will terminate the process immediately. If the access violates the page's memory protect, e.g., writing to a page marked as read-only, the kernel will send a `SIGSEGV` signal to the running process. When this signal is later processed, the kernel checks whether the process registered a signal handler. If it has, the handler will execute. If it has not, the default kernel signal handler will execute, which will terminate the process.

1.3 Interrupts, Exceptions, and Signals

Intel categorizes interruptions to normal execution into two groups: asynchronous and synchronous. Asynchronous interruptions, more typically known as interrupts, are raised by hardware device controllers. For example, a keyboard device will raise an interrupt when a user presses a key. These are unpredictable with respect to the execution of a program, as they are externally generated. Synchronous interruptions, also known as exceptions, are thrown by the CPU as the result of executing some instruction. These are synchronous with respect to the execution of a program as the CPU will always throw the same exception when executing the instruction that faulted. Page faults are an example of such exceptions.

Every interruption, synchronous or asynchronous, identifies with an interrupt vector. When the operating system is first booted, it will create a table saving all interrupt vectors with a corresponding handler to run when the interrupt is raised; these handler functions are traditionally part of kernel code. When an interrupt is raised by hardware, the CPU will automatically save the current application execution context and execute the registered interrupt handler. Assuming that the handler completes execution without raising another interrupt, the previous context will be restored and the application will continue.

This method of handling interruptions is appropriate when the interruption is intended to be handled by the operating system. However, there are several applications for when select interruptions are meant to be handled by the user code. As briefly mentioned in the previous section, user programs can install signal handlers that the kernel executes to process certain events. From the example in that section, if the user signal handler can change the page permissions for the illegal access, then the cause of the page fault will be resolved. When the faulting instruction re-executes, it will not fault again. If the interruption is intended to be handled by user code, there should be some way for that user code to be executed directly, without executing kernel code.

The end-to-end process of signal handling involves several privilege level switches between user space and kernel space. The current path of user-level signal handling involves the user application falling into kernel space code to set up the user handler trampoline, jumping to the user space handler, falling back to the kernel to restore the state pre-interrupt, and finally, resuming user program execution [Fig. 1.1]. The privilege level switches should be unnecessary for a fault purposefully raised by the user application to be handled by the user application. By removing the back-and-forth switching between kernel space and user space, the faulting mechanism will not invoke as many cache misses. This will decrease the overhead of handling a fault.

Instead of modifying the existing faulting pipeline and page fault handler, we create a new mechanism for a new type of page fault, a User Level Page Fault, to reduce the pipeline overhead for specific applications. In this thesis, we seek to imitate existing user signal handling behavior while completely bypassing any kernel interference. We will implement a page faulting mechanism that jumps directly to the user handler in user space. We use this mechanism for handling user-set memory protection violations and demonstrate its utility in a watchpoints library. We use the term "user-set" to describe that these permissions were added by the user process to the memory within its address space, in contrast to the permissions that the kernel set to its pages when the process is initialized.

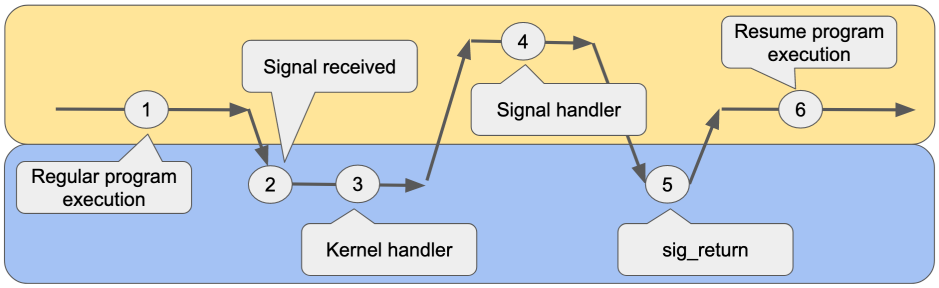

Figure 1.1: Signal Handling Pipeline

1.4 Thesis Statement

It is possible for the CPU to bypass executing kernel code to directly execute a handler in user code when a User Level Page Fault is triggered while still offering the same level of safety as existing architectures.

1.5 Roadmap

There is no mechanism in existing systems for an exception to fault directly into user space. This thesis offers a mechanism design for the CPU to directly execute user handler code when a program instruction violates its user-set permission. Our design delivers on three goals:

- Users can set and modify their user-set permissions without kernel involvement.
- Hardware enforces these user-set memory permissions.
- Exceptions raised by violating these user-set permissions will fault directly to user handler code, without kernel involvement.

This thesis has the following structure. Chapter 2 will provide a more general background on previous work on faulting to user space. Chapter 3 will explain, in detail, changes made to the hardware and software components. Chapter 4 will cover changes made to adapt our mechanism for a watchpoints application along with results from benchmarks. Chapter 5 will cover viable places for improvement that we did not have a chance to explore. Chapter 6 will mention some issues for future work to address. Chapter 7 will conclude our report and provide a general reflection and discussion about this mechanism. Finally, the appendix contains references that future developers may find useful in replicating or continuing our work.

Chapter 2

User-Level Exceptions

2.1 Problem Statement

In this project, we design a mechanism in which the CPU, upon some condition, raises a special exception that would execute a user-registered handler function and proceed with the rest of program execution. Our goal is to completely bypass any kernel interference, eliminating the overhead of jumping to the kernel, then from the kernel space exception handler, to the user space handler, back to the kernel space handler, and finally back to user space to resume the program. Since the fault was triggered by the user space program, we want to stay in user space to handle the exception and proceed with execution.

There are some existing frameworks that we build upon, mainly the existing interrupt/exception handling mechanism. There are already x86 instructions, `INT` and `IRET`, that can invoke the handler code and return to program execution. However, the registers that the CPU saves and restores when invoking the `INT` and `IRET` are not necessary for our use case. Also, the CPU is currently programmed to reference the interrupt descriptor table (IDT) to determine the function to execute when it raises an exception. While we initially considered just adding another entry in the table, we found that the table was set to read-only after the initial setup; it was designed to be set up once and not modified again. Overall, the exception handling mechanism mostly suits our needs; we need to create another channel to invoke it.

2.2 Previous Work

The overhead of switching back and forth between user space and the kernel incurs a non-negligible overhead on program performance. Invoking system calls, Soares et al. breaks down how executing kernel code impacts application runtime [10]. System calls, sometimes referred to as traps, are how users modify kernel-managed data. Traps and faults are both subclasses of exceptions and enter kernel space with near-identical mechanisms. Traps differ from faults in that the faulting instruction will be re-executed after it is handled whereas execution proceeds with the instruction following the trap instruction. The authors found that user mode IPC can degrade between 20-60% depending on the frequency of invoking system calls. They attribute this overhead to two factors: pipeline cost and pollution cost. The pipeline cost involves switching

between kernel and user space and flushing the processor pipeline. The pollution cost involves changes to the underlying architecture to run kernel code. This is mainly attributed to evictions in the L2, L3, and d-TLB caches that will create caches misses later in the execution [10].

There has been previous work done around moving kernel components to user space to reduce processing overhead. One area of work involves moving select device handlers to user level as the handlers handle hardware interrupts that might ultimately be handled by user level code. Caulfield et al. created a mechanim, Moneta-D [4], that provides a channel for transparently bypassing the kernel for processing I/O operations from their custom Moneta storage architecture [3]. Their user library maps the device control registers and device-writable memory into the address space of the calling process. The kernel stores the files' physical byte range and their respective permissions, which the hardware consults before serving I/O requests. Moving device drivers to user space also has been used to decrease kernel size for microkernel development. However, even if the interrupt handler code is in user space, the microkernel is still responsible for receiving the interrupt and redirecting it to a user task to process. That being said, the kernel's responsibility can be minimal. Upon receiving an interrupt in the Raven microkernel [8], the kernel performs a check to see whether kernel code handles the interrupt or user code handles the interrupt. If user code handles the interrupt, the kernel will pass the interrupt information to the user level dispatcher, which then invokes the actual handler. The device driver maps its register set into the application address space when it initializes. So, the user handler function can access device data without kernel involvement.

Chapter 3

User-Level Page Fault Design

The traditional architecture for exception handling involves components from the hardware and operating system. For the specific case of page fault exceptions, the major hardware components involved are the memory management unit (MMU) and the translation look-aside buffer (TLB). We will discuss these components in more detail later. The page tables and the exception handling mechanism within the operating system are the major software components. In this section, we will first explain the current architecture for page fault exception handling. We will then go over the modifications we made to each component of the architecture to implement User-Level Page Faults (ULPFs).

For the remainder of this report, variables proceeded by %, e.g. %AX, will represent the hardware register while AX will represent values stored in the register.

I will also refer to the set of following registers as "exception context registers": %SS, %RSP, %RFLAGS, %CS, and %RIP. The hardware automatically stores the values within these registers onto the kernel stack when an exception occurs. After executing the exception handler and prior to returning to user code, the hardware recovers these saved values from the kernel stack back to their corresponding registers.

3.1 Overview

Exceptions are software interrupts generated by either the hardware or the operating system upon some event. Exception handlers are installed into the interrupt descriptor table (IDT) when the operating system boots.

When an exception is triggered, the hardware replaces the user code's context registers with the exception handler's context registers in the following steps:

1. The CPU saves the values of major context registers (%SS, %RSP, %RFLAGS, %CS, and %RIP) to internal registers.
2. The CPU loads the kernel stack segment (SS0) and stack pointer (RSP0) values into %SS and %RSP, respectively. These values are stored within the Task State Segment (TSS) whose address is stored in the task register (%TR), both of which are written when the kernel boots.

Once the hardware registers %SS and %RSP hold SS0 and RSP0, all operations will be done in kernel space until the execution jumps back to user space.

3. The CPU pushes the previously saved context registers values (saved within internal registers in step 1) onto the kernel stack.
4. The hardware may sometimes push an error code onto the kernel stack, depending on what caused the exception.
5. From the IDT, the CPU finds and loads the kernel entry point and code segment offset for the exception into %RIP and %CS, respectively.

The kernel entry point pushes the execution context, e.g., all register values, onto the kernel stack and jump to the specific handler address for the exception.

In some cases, the exception handler will resolve the cause of the exception; these exceptions are known as faults. One example is when the hardware throws a page fault when accessing memory that is swapped out to disk. The kernel page fault exception handler will bring the page to memory. The execution unit will reexecute the faulting instruction and continue, because the memory access is now valid.

In other cases, the kernel exception handler will send a signal to the running process. If a signal handler exists, the kernel will save its current execution state in kernel space. The kernel then sets up a return stack on the user stack and executes the user signal handler in user space. After the user signal handler returns, the execution will be back in kernel space to restore the user execution context. Finally the CPU will once again jump back to user space to continue executing the original code. The overhead of signal handling is large, due to its repeated privilege-level switches between user space and kernel space. However, the privilege-level switches ensures that memory from one process is secure from other processes. Specifically, the program execution state must be saved within kernel space to be safe from other processes, which leads to the multiple privilege-level switches to save and restore this context.

3.1.1 ULI Architecture

This work on implementing ULPFs is built on an existing ULI (User Level Interrupts) architecture [11]. This architecture has a custom ULI hardware device specifically for delivering interrupts between different processors. Once a processor receives a ULI, the CPU raises a precise interrupt; the CPU saves the current program counter, flushes the execution pipeline, and fetches ULI microcode instructions. The ULI microcode saves the `RFLAGS` register value. It also saves the `RDI` register value, which holds the processor ID of the processor that sent the ULI. Afterwards, the CPU loads the ULI handler address, stored in a register we created, and executes the handler.

ULPFs are exceptions raised and handled on the same processor while ULIs are meant to be raised and handled between separate processors. However, we use the ULI pipeline and microcode to implement ULPFs.

3.1.2 Problem Statement

The x86_64 architecture and current operating systems does not lend itself to have low-overhead user-level, hardware-enforced memory protection for several reasons. First, the user cannot access its process page tables directly. Rather, the application must request the operating system to modify paging structures through system calls, such as `mprotect` and `mmap`, with specific permissions. Second, the current faulting architecture automatically jumps to a kernel address when a fault occurs.

If the problem is simply invoking a user handler at a permission violation, users can solve this problem by using the current signal handling architecture (Example code: 3.1). However, the current signal handling mechanism and having to invoke multiple system calls add 3000-4000 cycles of overhead per system call to the application.

```
void signal_handler(int sig, siginfo_t* info) {
  void* fault = info->si_addr;
  mprotect(mm_aligned, 4096, PROT_WRITE);
}


void setHandler(void (*handler)(int,siginfo_t *))
{
    struct sigaction action;
    action.sa_flags = SA_SIGINFO;
    action.sa_sigaction = handler;

    if (sigaction(SIGSEGV, &action, NULL) == -1) {
        perror("sig action err\n");
        _exit(1);
    }
}
int main(int argc, char**argv) {

    setHandler(signal_handler);
    char* mm_addr = malloc((PAGE_SIZE*2 - 1) * sizeof(char));
    mm_aligned = (char *)(((int) mm_addr + PAGE_SIZE-1) & ~(PAGE_SIZE-1));
    mprotect(mm_aligned, PAGE_SIZE, PROT_READ);

  // Regular program code
}
```

Listing 3.1: Signal Handling

Our goal is to decouple a permission violation page fault, which would send a signal to the user program, and a disk I/O page fault, which the kernel should handle. We separate these two cases into two different mechanisms. The kernel needs to handle disk I/O page faults ensure the safety of process data; this is the existing page fault mechanism. The second mechanism will specifically handle address permission violations in the context of the executing process; we optimize this mechanism to bypass all kernel intervention.

3.2 Hardware Modifications

The memory management unit (MMU) within the CPU performs the virtual-to-physical address translations and raises page fault exceptions. We modify this mechanism to implement ULPFs. The components we change include the virtual-to-physical address translation logic within the

MMU, the faulting mechanism microarchitecture, and we add registers to store data specific to ULPFs.

We first describe the existing faulting mechanism in detail. We then explain the mircoarchitectural changes to we made on the existing design to implement ULPFs.

3.2.1 Current Page Fault Handling

There are several hardware components the MMU and the operating system use to coordinate the paging system. The purpose of paging is to create a distinct address space for individual processes by creating an abstraction for the user, away from directly accessing physical memory. To accomplish this, the operating system holds a per-process page table (referred to as the kernel page tables) that holds the virtual address to physical address mappings. Kernel page tables are a hierarchical mapping structure between user-visible virtual memory addresses and hardware-visible physical memory addresses. Its structure is defined as part of the x86 architecture.

The hardware, specifically the MMU, traverses kernel page tables to translate virtual addresses to physical addresses. The hardware register %CR3 stores the base address of the page table for the current executing process. This structure is also known as the page global directory (PGD), is stored in the hardware register %CR3. The kernel is responsible for writing this register with the PGD of the running process during context switches. The memory permissions for each page are also stored within page table entries. If the MMU sees that the executing instruction will violate the permissions in the page table entries, the MMU will throw a page fault. When a page fault is thrown on an address, the MMU will load the address that caused the fault in to hardware register %CR2. The kernel page fault handler uses this value to diagnose the cause of the exception.

Address translation is a very mechanical process. The MMU takes a 48-bit virtual address as input and uses the address itself to index into the kernel page table structures. The 48-bit address is partitioned into five parts: the first 36 bits are partitioned into four 9-bit sections and the fifth part is the remaining 12 bits. The first 9 bits are used to index into the PGD. If we represent the value of the 9 bits as `i`, then we will retrieve the value stored in the `i`th element of the PGD. This value is the base address of the next level of the page table, also known as the page upper directory (PUD). We use the following 9-bit section from the virtual address to index into the PUD to get the next level of page table, or the page middle directory (PMD). Again, we use the following 9-bit section from the virtual address to get the next level of the page table, the page table entries (PTE). We use the last 9-bit section to index into the PTE, which will give the address of the physical page frame. Finally, we append the last 12 bits of the given virtual address to the physical page frame to get the physical address that corresponds to the virtual address. This process is illustrated in Figure 3.3.

For the purposes of implementing User Level Page Faults, which are very similar to the permission violation kernel page fault, we borrowed from the existing page faulting framework. We duplicated most of the same translation logic for traversing the ULPF page tables (these will be referred to as ULPTs); as such, most of the page table structure remained unchanged from current kernel page tables. The only difference is in the last level of tables, in which our design differs from current designs. This will be explained in more detail in section 3.4.

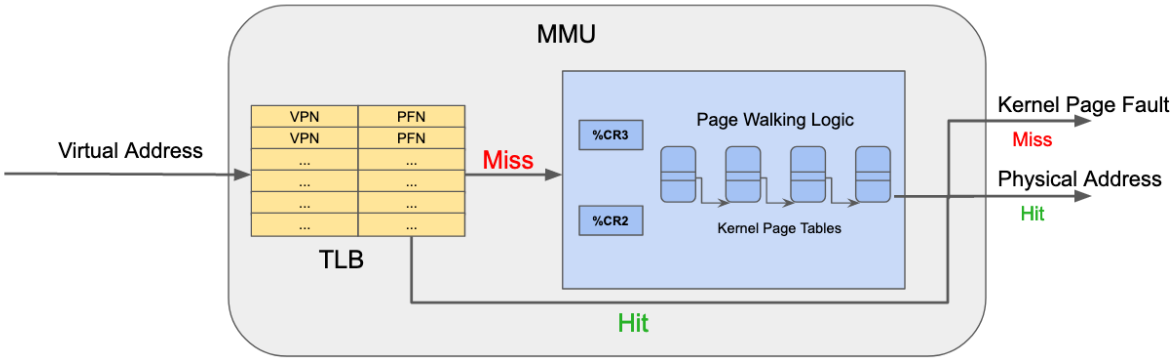

Figure 3.1: General memory translation logic flow

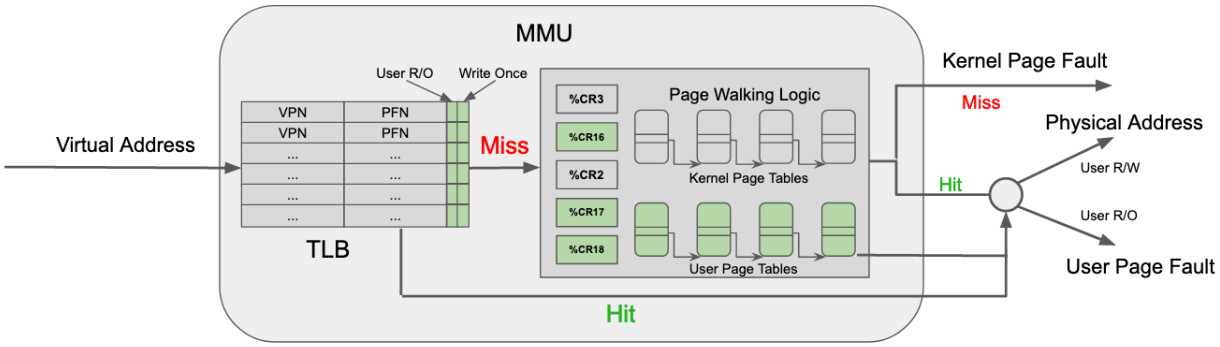

Figure 3.2: Hardware changes for ULPF

3.2.2 Modifications

In total, we add three registers, a write-once bit and user read-only bit in the TLB, and extend MMU logic to implement ULPFs.

Registers

We added three registers to support ULPFs that parallel the design in current architectures. %CR16 holds the base address of the user page table, analogous to how %CR3 holds the kernel process page table. %CR17 holds the value of the address that caused the ULPF, analogous to %CR2. %CR18 holds the address of the user-registered ULPF handler. The user page table base address and user handler address are saved within the process control block within the kernel and will be swapped in and out upon context switch.

Our first implementation registered the user handler as a new entry within the IDT to parallel current systems completely. However, this will allow only one application to use this mecha-

nism, because the IDT is typically written to once during boot time and set as read-only for the remainder of execution. Even if we took away this requirement, we would have to modify this IDT entry on every context switch, which increases the critical path. By holding the handler address in a new register, we increase the number of applications that can use this mechanism at the cost of a register write every context switch. We have also added additional x86 instructions that can read %CR17 and write %CR16, %CR18.

Page Walking Logic

In our design, the MMU will traverse both the kernel and user page tables concurrently (Fig. 3.3). This way, the addition of this mechanism will not incur an increased critical path length. Unlike walking the kernel page tables, walking the user page tables will never cause a fault for a nonexistent entry. If any mid-level entry in a kernel page table is absent, the MMU will throw either a page fault to bring data into memory or a segmentation fault. In contrast, if an entry is absent in the user-level page table, we silently abort any further page walking on the user page tables, while the kernel page walking will continue.

When a page of memory is allocated, the page is mapped within the kernel page tables. Our mechanism does not automatically map this page into the user page tables. It is unnecessary to map the page within the user page tables if the user never sets user-level permissions to this memory. We also do not want to force the user to set user-level permissions to all their allocated memory. To accommodate having empty user page table entries on potentially valid address translations, the MMU walks both the kernel and user tables with the following logic: If the kernel page table entry is invalid, the MMU raises a kernel page fault; if the user page table entry is invalid, the MMU continues page walking as it does in current architectures, only traversing the kernel page tables.

If there is an instance where a translation will cause both a kernel and user page fault, we raise the kernel handler.

TLB

TLB within Current Address Translation The TLB is a cache to store recent virtual to physical page translations. It takes advantage of locality as the translations for addresses on the same page are stored within the TLB. Having translations cached shortens the critical path for instruction execution as page-walking need not occur. Upon receiving a virtual address, the MMU will first search the TLB for an existing translation. If an entry exists, the physical address returns immediately. If one does not, then the MMU will perform the page walk, return the physical address, and cache the translation in a TLB entry.

TLB Permission Caching We made several modifications to the TLB design to accommodate our mechanism. We add a user-level-read-only bit within TLB entries that stores the user-level permissions. When the MMU performs a page walk on both the kernel and user-level page tables, the hardware stores the page frame number from the kernel page tables and the user level permissions from the user page tables within a TLB entry.

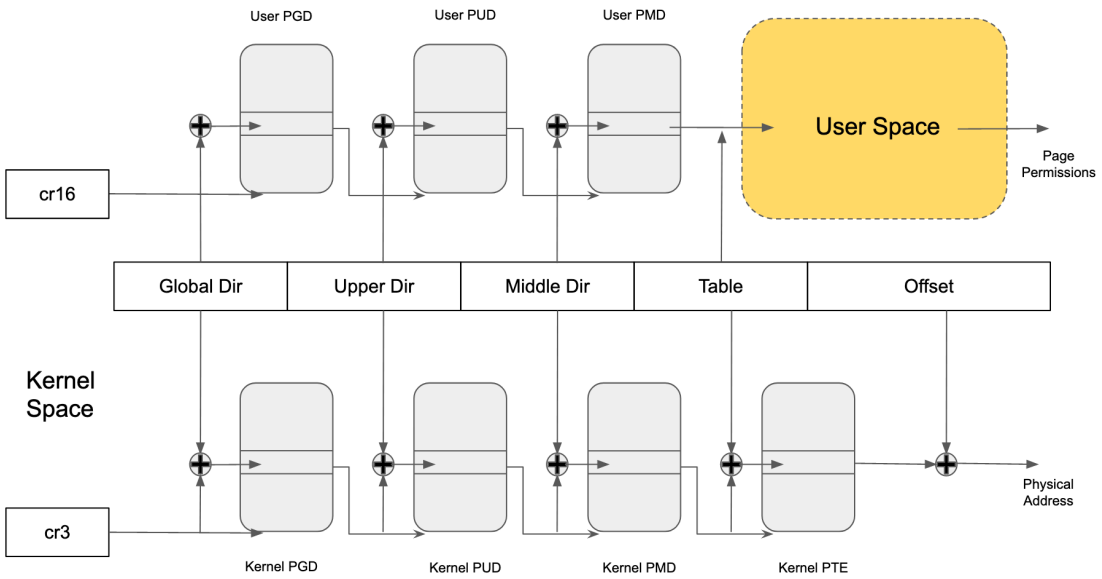

Figure 3.3: Address translation with kernel and user-level page tables

However, the MMU will not walk the page tables if the address translation is already cached within the TLB. As a concrete example, suppose the user allocated a block of memory and writes to an address in this memory. At this point, the translation for the accessed address is cached within a TLB entry. Suppose the user now sets a user-level read-only permission on this address. The next time the user writes to this address, there will not be a ULPF because the TLB entry is not updated with the new user-level permissions. The MMU will not traverse the user page tables to get the permission because there is already a cached TLB entry. Thus, we need to extend our mechanism to be able to update TLB entries to keep the ULPTs and TLB entries consistent.

So, if the address translation for a certain page is already cached with no user permissions and the user set it as read-only in the user page tables, the change will not be reflected in execution as the permission did not change in the TLB. To address this issue, we need a method to allow users to modify a TLB entry corresponding to a virtual page number from user space.

The `invlpg` x86 instruction is one of the only methods to change a TLB entry from software; it invalidates the TLB entry for a virtual page number and forces a page walk to occur. There are two issues with using this instruction for our purposes: the first is that this is a privileged instruction, the second is that we feel that a user-level operation should not be able to shootdown a system-controlled TLB entry, especially if used in a malicious program that can repeatedly shoot down entries with a single write in user space. While we had the option of creating a user-available `invlpg` to address the first point, we implemented a set of new `x86` instructions, `SET_TLB_RO` and `SET_TLB_RW`, which can be invoked from user space and can only toggle this additional user read-only bit we added in TLB entries. It is the user's responsibility to invoke these instructions to maintain coherence between the user page tables and the TLB. If the instructions are invoked with addressed not in the TLB, they will be 'NO-OPs.

Compiler Modifications and ULIRET It is important to mention the compiler modifications done to accommodate compiling user-level page fault handlers (credit to Chrisma Pahka, CMU). We defined a special attribute, `ulihandler __attribute__ (user_level_interrupt)` , to identify the user-level handler function to the compiler. We modified the compiler such that we insert a prologue and epilogue around the compiled handler function that saves and restores the user registers (e.g. %RCX, %RDX) that the compiled handler code uses. After the end of the epilogue, we insert a new x86 instruction, `ULIRET`, to return from a user level handler.

ULPT Faulting Pipeline Let us take a moment to tie all components together and go through the process in more detail. The MMU to translates a virtual address as follows:

1. The address will be checked against TLB entries to see if its translation is cached.
    - If it is, then its user permission bit would also be cached in the entry. If the entry is marked as user read-only and the operation is a memory write, then the CPU will execute the faulting pipeline.
2. If the entry is not in the TLB, the MMU will walk both the kernel page tables and user page tables at the same time.
    - If the entry is in both page tables, we check whether the access is legal within the kernel table entry.
        - If the access is illegal within the kernel page table, then we raise a kernel page fault.
        - If the access is legal within the kernel the page table, we check the permissions of the user page table, raising a ULPF is the access violates user permissions or returning the physical address if the access is legal.
3. If a ULPF is raised, the CPU will execute microcode that pushes the current %RFLAGS and %RIP onto the user stack.
4. The CPU will execute the user handler code. This will include: the prologue inserted by the compiler that pushes all registers used within the handler code, the actual handler code, the epilogue inserted by the compiler that restore the registers saved in the prologue, and the `ULIRET` instruction inserted by the compiler.
5. The `ULIRET` instruction runs microcode that restores the %RFLAGS and %RIP that were previously stored on the user stack by the hardware.

3.2.3 Instruction Re-execution

Assuming that the instruction successfully faults into the user space handler and returns, the hardware will attempt to execute the faulting instruction again. Reexcuting the instruction will involve going through the entire faulting process again, which would fault again if the address permissions are not updated to allow the execution. If the user does not update the address permissions to allow the write, this mechanism will loop forever. In addition, if the user modifies the permissions to allow this one write, the user must reset the permissions on the page for the next memory operation on the address to fault. We made the design decision to fault on the

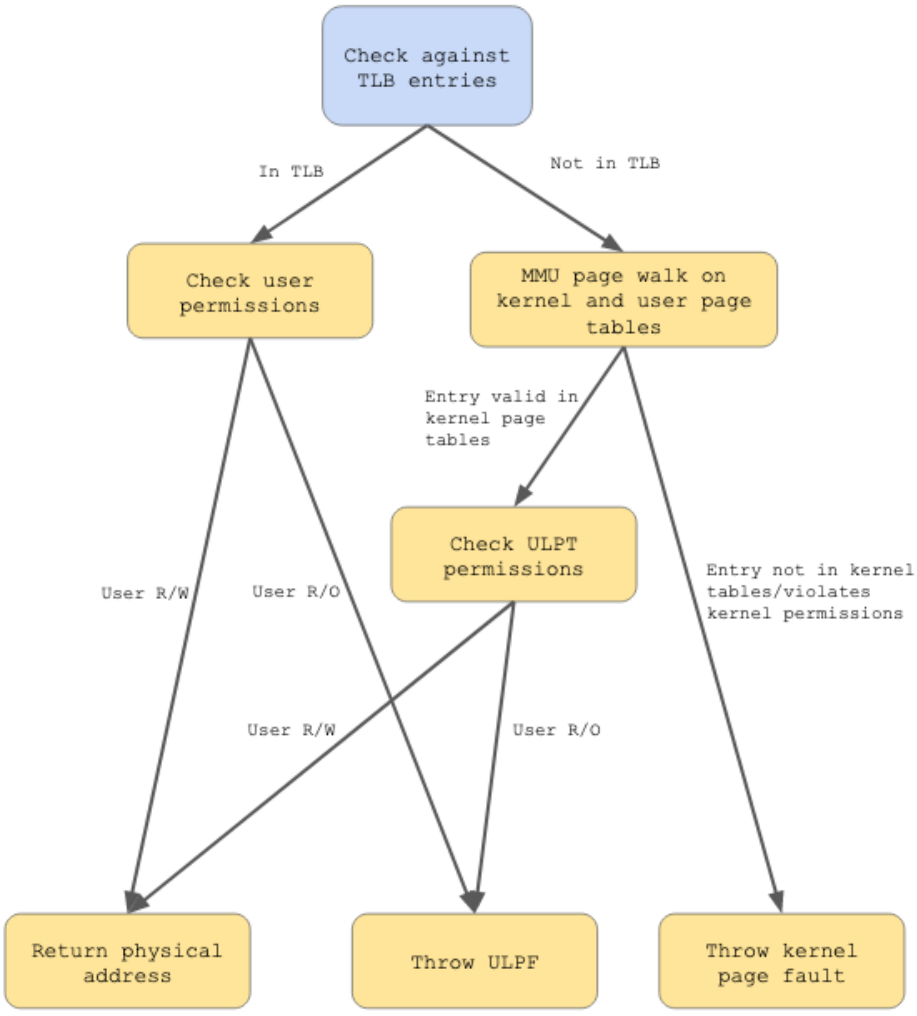

Figure 3.4: Logic diagram for checking for user-set permissions and raising a ULPF.

first memory operation on an address and automatically allow the second memory operation, the reexecution, to proceed without any user modifications.

We added a second bit, the write-once bit, within the TLB entry. The default state for this bit is 0, indicating that the current operation on this address is executing for the first time, and thus, should fault. Just prior to executing the fault handler, this bit is set. On the second execution, the MMU sees that this bit is set in the TLB and allows the instruction to execute without faulting. Right before executing the instruction, the TLB resets the write-once bit to 0 in the TLB entry so the next memory operation within the page will also fault.

This mechanism, as currently described, fails when executing CISC instructions that operate on separate memory addresses. One example of such a CISC instruction is `MOVUPS_XMM_XMM`, which breaks down into two move operations on different addresses.

```
1: movfp xmm_low, xmm_low_new
2: movfp xmm_high, xmm_high_new
```

The %RIP value for the instruction is `1`. The execution will be as follows:

1. The MMU translates the virtual address in the first microoperation.
2. The translation triggers the user-level handler; the write-once bit for the TLB entry for the address will be set.
3. The user level handler executes and returns, reloading the %RIP, or the first microp, again.
4. The MMU clears the write-once bit in the TLB and executes the instruction.
5. The MMU translates the virtual address in the second microoperation
6. The translation will trigger the user-level handler; the write-once bit for the TLB entry for the address will be set.
7. The user level handler will execute and return, reloading the %RIP, or `1`, again.
8. Steps 4-7 will loop indefinitely.

We add an internal register within the TLB to save the address used in the last memory operation. When the MMU translates a virtual address, it first checks whether the write-once bit is set. If the bit is not set, then the instruction has not faulted before. Here, we also save the faulting memory address in the internal register. If we see that the write-once bit is set, we know that we have already faulted on this instruction. However, this instruction may contain microcode that performs multiple memory references, so we check the current faulting address against the previous faulting address that we have saved. If the addresses don't match, we continue execution because we have not reached the microinstruction that we faulted on. If the addresses match, we continue execution but reset the write-once bit and clear the saved faulting address so the next microinstruction can fault. We are basing this logic on the assumption that microoperation sequences will not refer to the same memory instruction more than once. This logic is illustrated in Figure 3.5.

3.3 Software Modifications

We prioritized several goals in our design. First, we are looking for a paging structure that somewhat resembles existing structures to simplify necessary hardware changes. Second, we

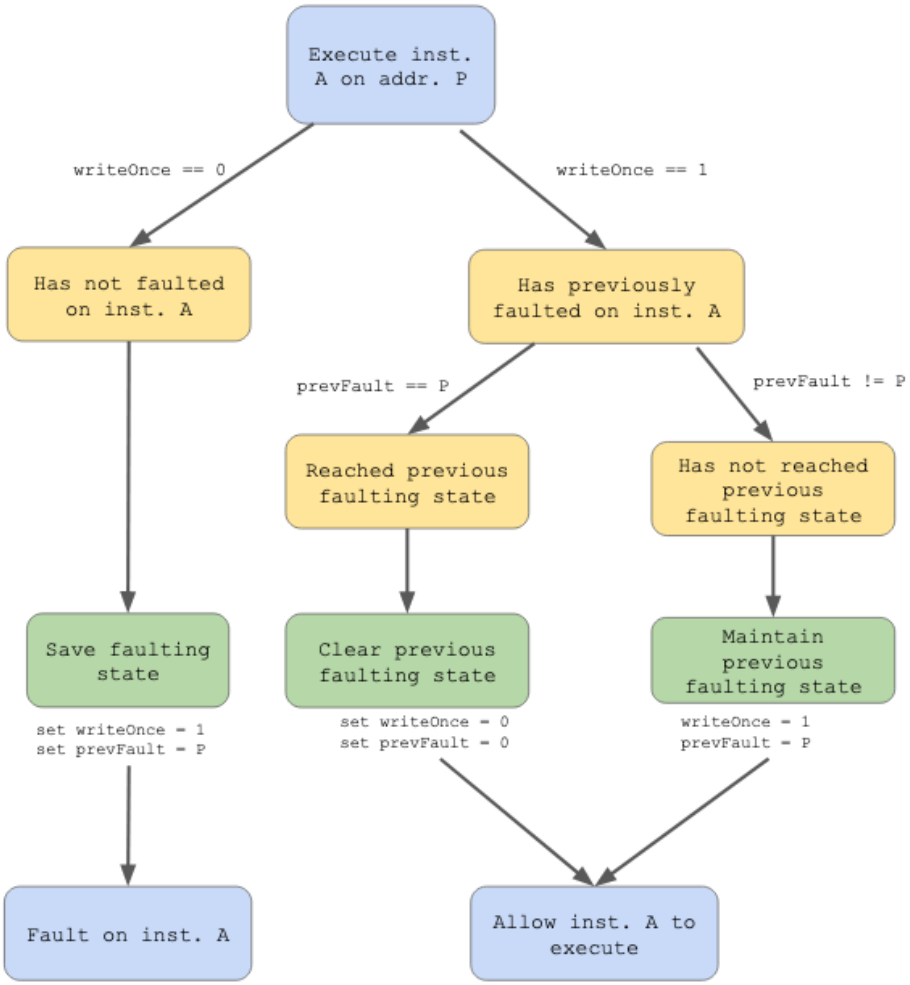

Figure 3.5: Logic diagram for the write-once mechanism. Yellow blocks are commentary; green blocks describe state; blue blocks describe execution.

would like the user to be able to modify the table on a page-granularity without invoking kernel code, meaning that the last level of the tables must be in user space. Finally, we would also like our structures to take up as little space as possible.

Given these requirements, we divided the software modifications into two major portions: the kernel portion and the user library. The kernel portion involves system calls that will install and manage the first three levels of the table in kernel space; the user library will handle the user interface and manage the user space level of the user page table.

3.3.1 Kernel Modifications

Current Design

The kernel is responsible for storing and managing page tables for every process. Page tables are typically copied from the parent process and their contents are replaced with the code and data of the new process. While there are multiple running processes, the kernel scheduler is responsible for context switching between them. Context switching involves writing the %CR3 register, which holds the base address of the current process' page global directory.

The only way for users to interact with system-level resources, like their page table, is to make system calls, e.g. `mprotect` and `mmap`.

All kernel modifications were applied to Linux version 2.6.22.9. To support ULPFs, we added two system calls and extended some existing process structures to support user-level interrupts.

Modifications

There are four levels to kernel page tables to accommodate 64-bit architectures. Since user-level page tables are a per-process structure, we would like the maximal number of table levels to be in kernel space. So, the first three levels of kernel page tables are duplicated to be the first three levels of user page tables.

The first system call, `long sys_register_uli(void *user_handler_addr)`, makes the hardware aware that the application has activated ULIs. The function will write to CR16 and CR18, respectively holding values for the user-level paging structure and the address of the user-level handler. The process' control block is extended to save these two addresses. The system call will also allocate the initial level of user level page table, the user PGD, to be in kernel space.

The second system call, `long sys_set_ro(void* vm_addr, void* addr_region)` updates the value corresponding to the PMD entry of the virtual address in the kernel page table to map to the given parameter, `addr_region`. The method to page walk the first two levels of the user-page tables is identical to that for the first two levels of the kernel page tables. Referring back to Figure 3.3, the given `vm_addr` will be a 48-bit virtual address. In address translation, this address is split into five discreet sections: the first 36 bits are split into four 9-bit sections that are used to index into intermediate page tables. The indexed entry stores the base address of the table used in the next level of the page walk. We copy this design exactly for the first two levels: the first 9 bits of the virtual address are used to index into the user PGD to get the base address of the user PUD. The next 9 bits are used to index into the user PMD to get the

base address of the user PMD. Here, the designs differ. The user PMD entry will hold the user-provided `addr_region`. Since this parameter is from user space, it is a virtual address. It must be translated to a physical address to be stored in the user PMD because the hardware assumes that table entries are physical addresses.

3.3.2 User Space Components

In contrast to the PTE of the kernel page tables, the fourth level of user level page tables is in user space; we refer to them as page level indicators (PLIs). A PLI is a page-aligned page of user-space memory that holds the user page permissions for a program using ULPFs.

We mentioned that we use the first 27 bit of the virtual address, that we call the virtual prefix, to index into the first three levels of the user page tables that are stored in kernel space. We are left with the remaining 9 bits of the page frame number (PFN) to index into a structure to get the page's permissions. A virtual prefix will correspond to a range of 512 consecutive pages. If we allocate two bits per page for permissions, we would have 1024 bits, or 128 bytes that would map to virtual addresses that have the same virtual prefix. In other words, one user PMD entry would only need to map to a 128-byte block; the remaining 9 bits of the virtual address can be used to index into this block to find the permissions for the page.

We partitioned the PLI into thirty-two, 128-byte regions; these regions are indexed from zero to thirty-one. Regions one through thirty-one on a PLI hold user-level page permissions for the current process; region zero is reserved for holding metadata about the structure.

Region zero holds three values. The first, `free_regions`, is a 32-bit bitmap that represents the allocation state of the remaining thirty-one regions. If the `n`th region is allocated, the `n`th bit in this value is set. The two remaining values in region zero are the `prev` and `next` pointers; PLIs are kept in a doubly-linked list structure. When users set user-level permissions on a memory range, we allocate PLI regions that have enough bits to cover this range. There are two cases to consider for PLI region allocation. If the lower and upper bound virtual addresses for the user memory range has the same virtual prefix (i.e., the first 27 bits of the virtual addresses are the same), then one region can cover the memory range. If the lower and upper bound do not have the same virtual prefix, the numbers of regions to allocate is the range modular 4096, the number of pages one region covers.

We allocate regions by searching existing PLI structures to see if there are empty regions. If any bit in `free_regions` is not set, there is a free region in the current PLI. If the all the regions are allocated, we load the `next` pointer, which is the next PLI in the linked-list, and we check for free regions. If all regions in all allocated PLIs are allocated, we allocate a new PLI page and append it to the front of the PLI linked list. We then allocate the regions we need from this new PLI. Once a new region is allocated, it is passed as an argument to the `sys_set_ro` system call to be stored in the page's user PMD entry.

3.4 User Library Design

The ULPF library manages two key structures in user space: the PLI structures and the region lookup table. We described the PLIs earlier; they are the last level of the ULPTs that hold the

page permissions. As mentioned previously, the PLIs are pages partitioned into 32 regions where each region hold permissions to pages that map to the same virtual prefix. The ULPF library code is responsible for checking for free regions in existing PLIs and allocating a new PLI page if there are no free regions.

The region lookup table is a structure that stores the mappings between virtual address prefixes and the base address of the corresponding PLI region. The table is initialized as one page that holds 255 table entries. A table entry consists of two 64-bit values: the virtual prefix and the PLI region base address. When all entries are full, we allocate another page for the region lookup table and store the address of that page as the final entry of our existing region table. Thus, the region lookup table is also a linked list. These tables store entries as they are set by the user, so the entries are not sorted. We implement the `check_virt_prefix` function within the library to traverse the region tables to checked whether the virtual prefix of the given address is mapped in the ULPTs, i.e. has an entry in the region lookup table.

We use the region lookup table for two reasons. When users set user-level permissions on an address, we first check whether the virtual prefix is in the region lookup table. If it is, then setting user permissions will only require modifications in user level code. If the virtual prefix is not in the region lookup table, we know that the address is not mapped in the ULPTs in kernel space. Here, we must invoke the system call `sys_set_ro` to populate the ULPTs in kernel space before continuing with operations in user space.

Second, the return value of `check_virt_prefix` is the base address of the PLI region of the virtual prefix of the virtual address, if the address is mapped. This value is one of the input parameters for `modify_pli`, which is responsible for updating the permission bits for specific pages in the PLI structures. The signature of the function is `void modify_pli(void* vm_addr, int perms, unsigned long pli_region)`. `vm_addr` is the virtual address that on which user sets user permissions. `perms` are the user permissions; currently we only support `USER_RW` and `USER_RO`. The address, `pli_region`, is the base address of the PLI region returned from `check_virt_prefix`.

The actual bit-manipulation is as follows:

```
#define CHAR_BITS 0x7
#define OFFSET_SIZE 0x1FF

void modify_pli(void* vm_addr, int perms, unsigned long pli_region) {
    // Gets the nine bits that typically corresponds to the PTE index and
    // doubles it because there are two permission bits per page
    unsigned long region_offset = (((unsigned long)vm_addr >> 12) & OFFSET_SIZE) << 1;

    // Divide by 8 to get the byte within the region that the offset corresponds to
    char* region_byte = (char*)pli_region + (region_offset >> 3);

    // Mask to only get the last three bits to get the bit to flip within
    // region_byte
    unsigned char page_bit_first = (region_offset & CHAR_BITS);

    // BITAND to set the "page_bit"-th bit in region_byte
    switch (perms) {
    case USER_RW:
      *region_byte = (*region_byte & (~(1 << page_bit_first)));
      break;
    case USER_RO:
      *region_byte = (*region_byte | (1 << page_bit_first));
      break;
    default:
```

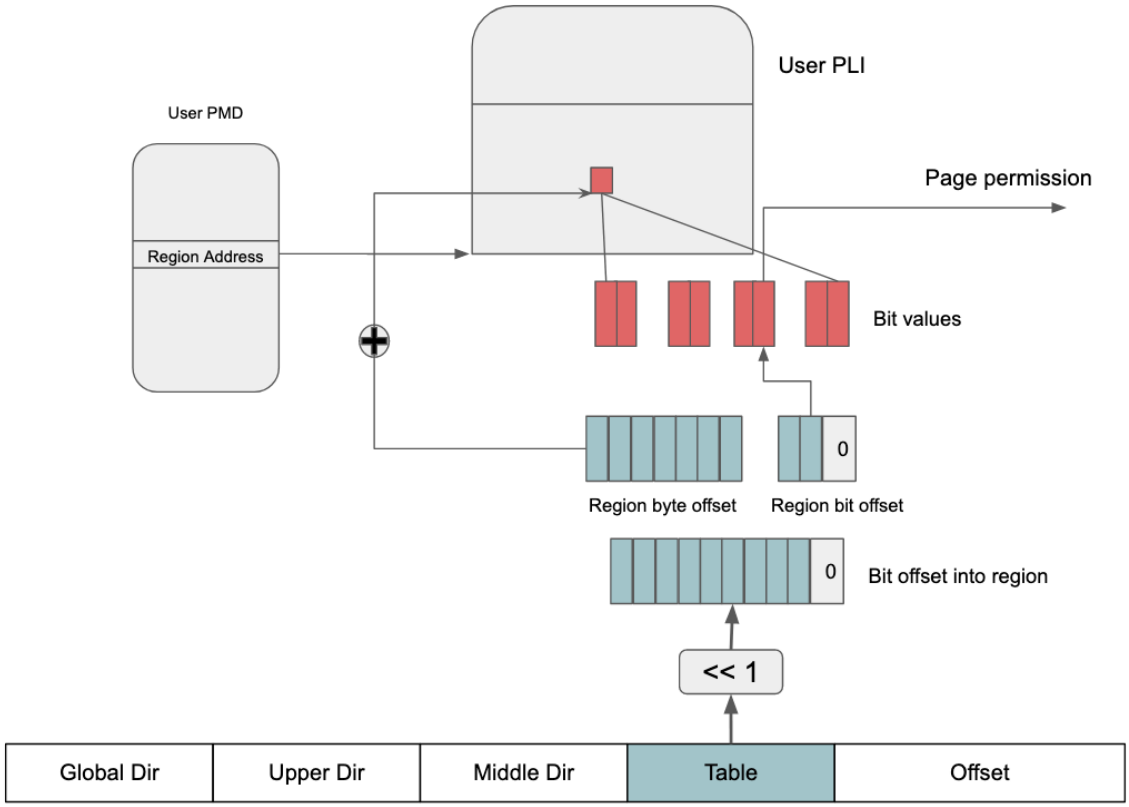

Figure 3.6: Indexing into the PLI using bits 12-20 of the virtual address to get the user-set permissions for the page. Bits 21-48 are used to find the corresponding user PMD structure.

```
25        printf("NOT VALID PERMISSION\n");
26    }
27      return;
28  }
```

Once the user changes the permissions pertaining to any address, we update the TLB to ensure coherence between the user page table structures and the hardware. Here, we invoke the new x86 instructions mentioned in the TLB section, `SET_TLB_RO` and `SET_TLB_RW`. If these instructions are called on an address not in the TLB, the instruction performs a NO-OP. This is illustrated in Figure 3.6.

3.5 Summary

To realize our design goals for ULPFs, we modify both hardware and software components. We ensure that ULPT permissions will be hardware enforced by basing our design off of existing page walking logic within the MMU. Likewise on the software front, we duplicate the kernel level page table structures for ULPTs to interface with the hardware extensions we made. We move the last level of ULPTs to user space so users can modify their memory permissions with a single write instruction without changing privilege levels. We bypass any kernel intervention

by modifying the interrupt pipeline to jump to user handler code immediately after the hardware throws a ULPF.

3.6 List of New Instructions, System Calls, and User API

```
/** Returns from the ULPF handler; restores %RFLAGS and %RIP that were saved by hardware **/
ULIRET

/** Sets the TLB entry for vaddr to be user read-only if it is within the TLB, NO-OP the entry
    does not exist **/
SET_TLB_RO (vaddr)

/** Sets the TLB entry for vaddr to be user read-write if it is within the TLB, NO-OP the entry
     does not exist **/
SET_TLB_RW (vaddr)

/** Writes the value vaddr into CR16 (the address of the user PGD) **/
WRITE_CR16(vaddr)
/** Reads the value stored in CR17 (the faulting address) and writes it to vaddr **/
READ_CR17(vaddr)
/** Writes the value vaddr into CR18 (the address of the ULPF handler) **/
WRITE_CR18(vaddr)
```

Listing 3.2: New ULPF x86 Instructions

```
/** This is not invoked directly by the user program, but rather through the user library.
    Initializes the User PGD level and writes address to CR16. Writes the user_handler_address
    into CR18 **/
long sys_register_uli( void* user_handler_addr);

/** This is not invoked directly by the user program, but rather through the user library. The
    library will manage the PLI structures that will designate the addr_region argument. The
    vm_addr is the virtual address whose ULPT entry will be set to read-only. **/
long sys_set_ro(void* vm_addr, void* addr_region)
```

Listing 3.3: New System Calls

```
/** Possible values for the PERMS value **/
#define USER_RW     0x0
#define USER_RO     0x1
#define USER_EXE    0x10

/** Wrapper around the sys_register_uli system call; initializes the user library structures (
    the virtual prefix map, a page for PLI, initialize the watchlist tree structure.)
int register_user_handler(void* handler_addr);

 /** Wrapper around the sys_set_ro system call. First checks whether the address range covered
    by [addr, addr + range) is already watched, checked via the user space region table. If the
     range is watched, the function returns. If the range is not entirely watched, the function
     will allocate a new region within a PLI and make the sys_set_ro system call with the new
    region address and the given addr. Adds a new node within the watchlist tree, merging
    existing nodes if necessary. **/
long set_permissions(void* addr, size_t range, int PERMS);

/** Wrapper around the READ_CR17 instruction.  Returns the virtual address stored in CR17,
    which holds the faulting address. **/
unsigned long read_fault_addr();

/** Searches through the watchlist splay tree to check whether the faulting address is being
    watched. Internally invokes the read_fault_addr function to find the faulting address. **/
int lookup_wp();
```

Listing 3.4: User API

Chapter 4

Case Study: Watchpoints

We use ULPFs to implement watchpoints that users can set within their program. We chose this application for our case study because watchpoints are one example where the hardware throws a fault for the running process that eventually gets handled in user code of the running process. A watchpoint is a debugging mechanism that suspends program execution when a "watched" address is modified. We implement watchpoints by setting the watched addresses to have read-only user-level permissions. When the CPU raises a ULPF, it will execute a user-level handler set by the user.

However, our ULPF mechanism stores permissions only on the page-level granularity. Users typically set watchpoints on individual addresses or small address ranges. Our mechanism will raise a ULPF for addresses within the same page as a watched address; we refer to these as false positives. To be able to differentiate between false positive addresses and true watched addresses, we built on our ULPF library with additional data structures to track watched addresses.

4.1 Watchpoint User Library

Our watchpoint user library provides users with a simple interface to set watchpoints within their program. It internally implements a *watchlist*, a structure containing watched address, to differentiate between true watched addresses and false positives.

All watched addresses are kept in a splay tree, sorted by address; this is our watchlist data structure. The ULPTs only store permissions pertaining to pages rather than individual addresses. Watchpoints, however, can be set on individual addresses. Using the unextended ULPF library, setting one address to have user read-only permissions is equivalent to setting the permission on the entire page. Thus, there needs to be a structure to track addresses on a finer granularity to distinguish actual watched addresses from false positives.

After some experimentation, we settled on using a splay tree as our watchlist structure. A splay tree is a self-balancing binary search tree that rotates the most recently accessed element to the root. This leads to having a smaller search time for elements that are accessed often. Being a binary search tree, every subtree holds the invariant that all elements to the right of the root will be larger than the root element and all elements to the left of the root will be smaller than the root. For our watchlist implementation, an element within the tree is an address range, defined

by a lower and upper bound. The library splay tree will dynamically merge overlapping address ranges as the user inserts them. This preserves our binary tree invariant and minimizes the tree size.

We provide three functions within our watchpoint user library:

- `long set_permissions(void* addr, size_t range, int PERMS)`
- `unsigned long read_fault_addr()`
- `int lookup_wp()`

We will explain these functions in more detail.

The first, `set_permissions(void* addr, size_t range, int PERMS)`, sets the address range from `[address, address + range)` to have `PERMS` permssions. In other words, this function sets the addresses within the given range to have user-level permissions. We assume that the user already initialized the ULPF mechanism prior to invoking this function. The ULPF mechanism is initialized by invoking the `register_user_handler` library call that we covered previously. The `set_permissions(void* addr, size_t range, int PERMS)` watchpoint library call is a wrapper function around internal ULPF library calls. First, it checks whether the virtual prefix of the virtual address is already mapped in the user page tables by invoking `check_virt_prefix`. If it is, then the function returns the base of the PLI region that maps to the virtual prefix. If it is not, then the `set_permissions` function allocates a new PLI region and invokes the `sys_set_ro` system call to map the virtual prefix into the ULPTs that reside in kernel space. Lastly, it invokes `modify_pli`, which actually find the permission bits in the PLI that corresponds to the given virtual address. The functions mentioned were covered in detail in Section 3.4.

The second function, `unsigned long read_fault_addr()`, reads the %CR17 register to find the address that caused the CPU to raise a ULPF.

The third function, `int lookup_wp()`, invokes `read_fault_addr` to find that address that caused the ULPF, and uses it to traverse the watchlist tree. This function returns zero if the address that faulted was not a watched address, i.e. a false positive, and returns one if the address that faulted is being watched.

Bringing everything together, a typical user application will look as such. The user would first register a ULPF handler function through the `register_user_handler` call, which initializes the user level paging in kernel space. The library call also initializes the PLI structures in user space. The user handler function will first check whether the faulting address is being watched by invoking `lookup_wp`. If it is a watched address, it will continue executing the handler. Otherwise, it will immediately return. The user can then set read-only permissions on individual addresses or ranges of addresses that are in its address space through `set_permissions` within the program.

```
1 /** Possible values for the PERMS value **/
2 #define USER_RW     0x0
3 #define USER_RO     0x1
4 #define USER_EXE    0x10
5
6 /** Wrapper around the sys_register_uli system call; initializes the user library structures (
      the virtual prefix map, a page for PLI, initialize the watchlist tree structure.)
7 int register_user_handler(void* handler_addr);
8
9  /** Wrapper around the sys_set_ro system call. First checks whether the address range covered
```

```
    by [addr, addr + range) is already watched, checked via the user space region map. If the
    range is watched, the function returns. If the range is not entirely watched, the function
    will allocate a new region within a PLI and make the sys_set_ro system call with the new
    region address and the given addr. Adds a new node within the watchlist tree, merging
    existing nodes if necessary. **/
long set_permissions(void* addr, size_t range, int PERMS);

/** Wrapper around the READ_CR17 instruction.  Returns the virtual address stored in CR17,
    which holds the faulting address. **/
unsigned long read_fault_addr();

/** Searches through the watchlist splay tree to check whether the faulting address is being
    watched. Internally invokes the read_fault_addr function to find the faulting address. **/
int lookup_wp();
```

Listing 4.1: Watchpoints API and Definitions

```

#define ulihandler __attribute__ ((user_level_interrupt)) \\
__attribute__ ((disable_tail_calls)) void
ulihandler user_handler() {
    void* addr = read_fault_addr();
    int found = lookup_wp();
    if (found) {
        printf("Watchpoint hit")
    }
    else {
        printf("False positive")
    }
}

int main(int argc, char**argv) {

    int ret = register_user_handler(&user_handler);
    if (ret < 0) {
        return -1;
    }

    char* mm_addr = malloc(PAGE_SIZE * sizeof(char));

    set_permissions(mm_addr, PAGE_SIZE, USER_RO);

    // Regular program code
}
```

Listing 4.2: Example Watchpoint Usage

4.2 Previous Work

There are several existing projects that explore watchpoint implementation. The first is GDB, a popular debugger. GDB utilizes the hardware debug registers for the first three watchpoints and falls back on a software watchpoint implementation if the user sets more watchpoints. GDB's internal software watchpoint implementation is very slow, as it forks the application as a child process and uses the `ptrace` system call to single step the application code and check whether the accessed address is watched against some internal data structure. The `ptrace` call also uses signals to communicate between the parent debugger and child application, which adds to the overall slowdown. While this enables setting any number of watchpoints, software watchpoints incur a huge slowdown (up to 800,000x) over the native application.

Another method to implement watchpoints is using dynamic binary instrumentation (DBI) along with shadow memory to execute a check on each memory access. DBI frameworks are used as program profilers that can observe behavior at runtime. Pin [5], one DBI framework, takes an executable file as input and rewrites the given binary using its internal just-in-time compiler. This allows Pin to inject user-installed monitoring code before each memory operation.

Shadow memory is specially allocated memory used to identify permissions for another region of memory. A naive implementation partitions the address space in two, with one region being program-accessible and the other being a one-to-one mapping for the permissions of bytes being used. Although space-inefficient, the implementation is straightforward; the user-installed instrumentation code would add a set offset to the accessed memory address to find its corresponding permission in the shadow memory and execute some routine. EDDI [13] uses DBI and shadow memory to efficiently implement unlimited watchpoints in programs running on IA32 architecture. They further decrease the overhead of their mechanism by shadowing one byte of memory with one bit, performing register analysis to reduce spilling when executing instrumentation code, and batch checking memory accesses [7].

Along a similar vein, MemTrace [6] expanded on a cross-ISA binary translator to provide memory watching capabilities. It cross compiles between 32-bit and 64-bit architectures and uses the additional registers and increased memory space to inject and execute small code sequences within the original executable. The concept is similar to that of EDDI but MemTrace used a binary translator, instead of a DBI, for code insertion. MemTrace also uses shadow memory to save permissions pertaining to the original 32-bit application's memory space.

The Mondrian Memory Protection (MMP) [12] scheme is one implementation that modifies hardware rather than only modifying software tools. MMP stores permission information corresponding to sections of user space memory within kernel space. A hardware component checks every address against this kernel structure. MMP and ULPF contain many similar components. We differ in our methods to update user permssions: MMP invokes an inter-protection domain call, ULPF directly writes to the PLI permission bit. When a permission violation triggers a fault, MMP invokes the traditional kernel fault, while ULPF faults to a user-defined handler. Our faulting mechanism has a lower overhead over that the inter-protection domain call that MMP uses.

4.3 Implementation Details

As we explained in the previous chapter, we needed to modify the hardware, the operating system, and add a user library. Our hardware modifications are simulated on Gem5 [1], an open-source computer architecture simulator. It also has a "full-system" emulation setting where it can emulate an unmodified kernel binary. Within its source code, we modified the page walking and TLB execution logic and implemented the additional aforementioned x86 instructions.

To invoke our new instructions, we also modified the LLVM compiler to compile ULPF programs. The compiler detects the user handler function through a unique attribute and will automatically generate assembly to save and restore the execution registers prior to and after executing the handler code. For example, if the compiler used registers %RAX, %RBX, and %RCX within the handler code, the final generated assembly code for the handler will be:

```
push %RAX
push %RBX
push %RCX
// handler code assembly
pop %RCX
pop %RBX
pop %RAX
```

We chose to modify Linux kernel 2.6. Unfortunately, we later found that Gem5 was unable to run some executables because the kernel version was too old.

For our benchmarks, we wrote programs in C and compiled with Clang 6.0 with our modified LLVM back-end. We compiled everything with `-O3` unless stated otherwise. We also found that some functions were not supported by the internal `libc` library within Gem5 so all our programs were statically linked against `glib 2.17`. Our cycle counts were taken using the `rdtsc` instruction with the start point after initial variable declaration and argument checking and the end point before any resource cleanup. Our reported cycle counts are the difference between the end point and the start point.

We ran benchmarks on two different user handler configurations. The first is an empty handler; this measures the overhead of the faulting mechanism. The second is a watchpoint handler. Upon every invocation of the user handler, it first invokes our library call to check whether the faulting address is a watched address. If it is, then it will increment a global `watchpoint_hit` variable by one; otherwise it will increment a global `false_positive` variable by one.

4.4 Comparison to Signal Handling

Table 4.1: ULPF to Signal Handling Comparison

	ULPF (cycles)	**Signal Handling (cycles)**
(1) Fault to entering handler	130	8800
(2) Handler lookup	200	210
(3) Allowing the write inst.	–	3500
(4) Exiting handler	85	5200
(5) Reexecute write	–	3600
(6) Restore R/O permissions	–	3600

When comparing our faulting mechanism as a faster signal handling mechanism, we found that in a step-by-step breakdown of that our permission-setting and faulting mechanism is up to 100x faster (Table 4.1) than the same a program of the same functionality written using signals (Listing 4.3). An important difference is that our architecture modifications are specifically targeted for this application. As a byproduct, steps 3, 5, and 6 (see Fig. 4.1) of the execution put regular signal handling at a disadvantage because our architecture handles the write-once operation within hardware. Other than that, there are major differences in comparing the cycles it take to point of faulting to the point that the handler code runs. The ULPF mechanism pushes only a couple of registers onto the user stack and jumps to the handler address. The signal handling mechanism would have to perform a stack change, execute the kernel handler, and then

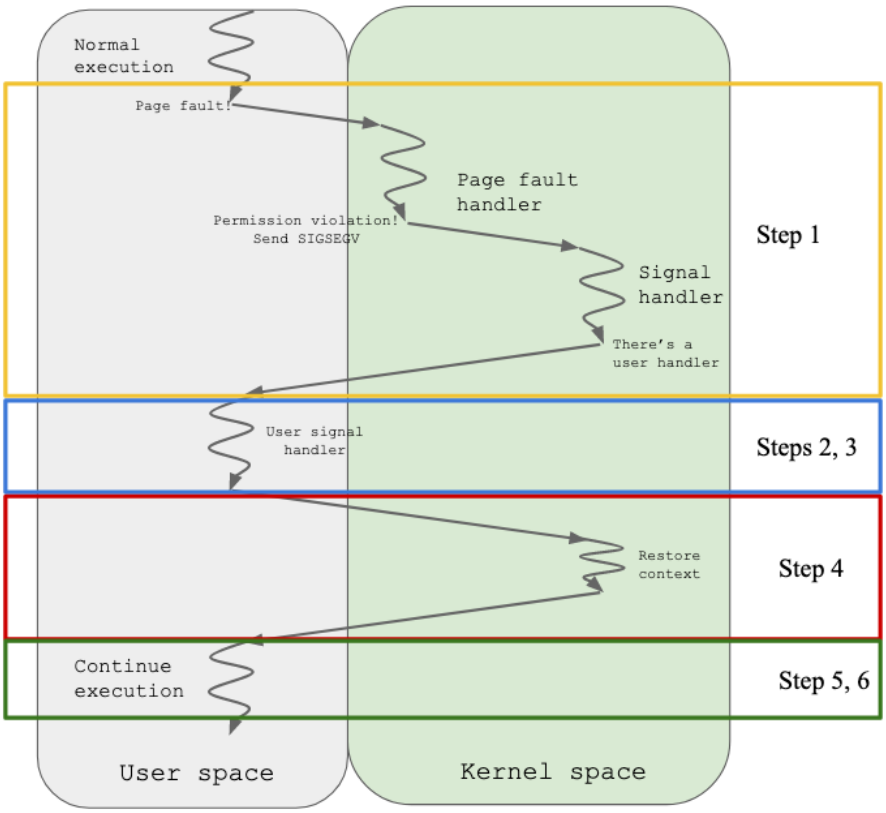

Figure 4.1: Signal execution steps corresponding to Table 4.1

jump to the signal handler address (step 1). Likewise, when we are returning from the handler, the ULPF approach uses a specialized instruction to return; it pops the saved registers and jumps to the faulting instruction address. The signal handling pipeline involves jumping back to kernel space to restore the original context, and then jumping back to the user program (step 4).

```
void signal_handler(int sig, siginfo_t* info) {
  void* fault = info->si_addr;
  mprotect(mm_aligned, 4096, PROT_WRITE);
}


void setHandler(void (*handler)(int,siginfo_t *))
{
    struct sigaction action;
    action.sa_flags = SA_SIGINFO;
    action.sa_sigaction = handler;

    if (sigaction(SIGSEGV, &action, NULL) == -1) {
        perror("sig action err\n");
        _exit(1);
    }
}
int main(int argc, char**argv) {

    setHandler(signal_handler);
    char* mm_addr = malloc((PAGE_SIZE*2 - 1) * sizeof(char));
    mm_aligned = (char *)(((int) mm_addr + PAGE_SIZE-1) & ~(PAGE_SIZE-1));
    mprotect(mm_aligned, PAGE_SIZE, PROT_READ);

  // Regular program code
}
```

Listing 4.3: Signal Handling

We further break down the cycles for the process from the time the CPU throws a ULPF to the point of executing the first instruction in the handler prologue (Table 4.2). In the future, we can potentially cut down cycles by saving multiple registers in parallel.

Table 4.2: ULPF Fault to handler breakdown

	ULPF (cycles)
(1) Flushing the pipeline	10
(2) Inform microarchitecture of ULI	6
(3) Save the next PC	6
(4) Save the current RFLAGS	12
(5) Save the current RDI register	12
(6) Write the CPU id to RDI	7
(7) Misprediction	10
(8) Execute first instruction of the handler prologue	2
Total	**65**

4.5 Microbenchmarks

Our first microbenchmark is a general overview of the scalability and overhead of a user-level page fault. Our test program allocates 256 consecutive pages, each 4096 bytes, and sets watchpoints on addresses according to different distributions. Our measurement times a loop that writes to each address in order. We also took measurements with two different user handlers: an empty handler and one that searches the watchlist tree to determine whether the faulting address is watched.

Table 4.3: Sparsely Distributed Watchpoints

WPs	Pages Spanned	False Positives	Slowdown (lookup)	Slowdown (no lookup)
1	1	4095	1.05	1.01
2	2	8190	1.11	1.02
3	3	12285	1.17	1.02
4	4	16380	1.22	1.03
8	8	32760	1.45	1.06
16	16	65520	1.91	1.13
32	32	131040	2.84	1.26
64	55	225216	4.16	1.45
128	101	413568	6.81	1.83
256	156	638720	9.97	2.29

Table 4.4: Densely Distributed Watchpoints

WPs	Pages Spanned	Slowdown (lookup)	Slowdown (no lookup)
8	1	1.05	1.01
16	1	1.05	1.01
24	2	1.11	1.01
32	2	1.11	1.01
40	3	1.16	1.02
48	3	1.16	1.02

In our first microbenchmark, we place watchpoints on uniformly distributed random addresses. We refer to this as the sparse configuration. From the data in Table 4.3, we see that the overall slowdown of the application is linear with respect to the number of pages spanned when there is no lookup in the user handler. It is also linear with respect to the number of watchpoints when invoking an empty handler (Fig. 4.2).

Our second microbenchmark watches addresses, also in one byte units, uniformly spaced 256 bytes apart. We refer to this as the dense configuration. Setting an address to be watched involves setting the entire page to be user read-only. This implies that even watching one address will lead to 4096 faults, of which 4095 are false positives. The difference between data in Table 4.3 and Table 4.4 reflects that the slowdown is independent of the number of addresses watched within the same number of pages spanned. Since the addresses watched in the dense configuration are

not consecutive, the number of nodes in the search tree is also equal to the number of watchpoints. However, the slowdown in Table 4.3 for 8 watchpoints is much higher than for 8 watchpoints in Table 4.4, because the benchmark in Table 4.3 triggered 8 times as many faults.

We had two additional microbenchmarks. One watches one address on each page and one watches the entire block of allocated memory. Both configurations will have the same number of pages spanned, leading to the same number of faults. Here, we see a slight variation in slowdown for just the mechanism when there are many faults, as shown in the Slowdown (no lookup) column of Table 4.5. However, we also see that the memory accesses from searching the watchlist accumulates even if there is only one node, as seen in the dense configuration. The slowdown for the sparse configuration may be optimistic, for we watch the first address on each page and insert them into the splay tree in the same order as our access pattern. This will lead to a fewer number of rotations when accessing the nodes of the splay tree.

Table 4.5: Extreme Distribution of Watchpoints

WPs	Pages Spanned	False Positives	Slowdown (lookup)	Slowdown (no lookup)
256	256	1048320	15.8	3.2
1048576	256	0	14.5	3.2

We ran the same microbenchmarks using GDB hardware and software watchpoints. In order to have a reasonable runtime, we reduce the overall allocated memory to 8 consecutive pages. We randomly select a number of addresses to watch in both GDB and user-level page faults. We then time the number of cycles it takes to write to all allocated bytes. For implementation details in GDB, please refer to Appendix A.

Microbenchmarks were compiled with `-O0` optimization so that GDB will be able to set watchpoints properly without variables being optimized out. This explains the ULPT slowdown being drastically larger than the slowdowns presented in earlier experiments. GDB only allows setting 3 hardware watchpoints at a time; the overall slowdown to using hardware watchpoints, e.g. the debug registers, is close to negligible. However, if users want to watch more than 3 discrete addresses, GDB software watchpoints will incur a large slowdown, close to a million times the original runtime.

Table 4.6: GDB Comparison (ULPF benchmark compiled with `-O0`)

Addresses Watched	GDB HW Slowdown	GDB SW Slowdown	ULPF Slowdown
1	1.05	799600	2.5
2	1.05	1018000	4.2
3	1.03	1203000	6.0
4	–	1446000	6.1
8	–	2101000	8.2
16	–	3773000	12.0
32	–	6546000	17.1
64	–	13520000	19.4

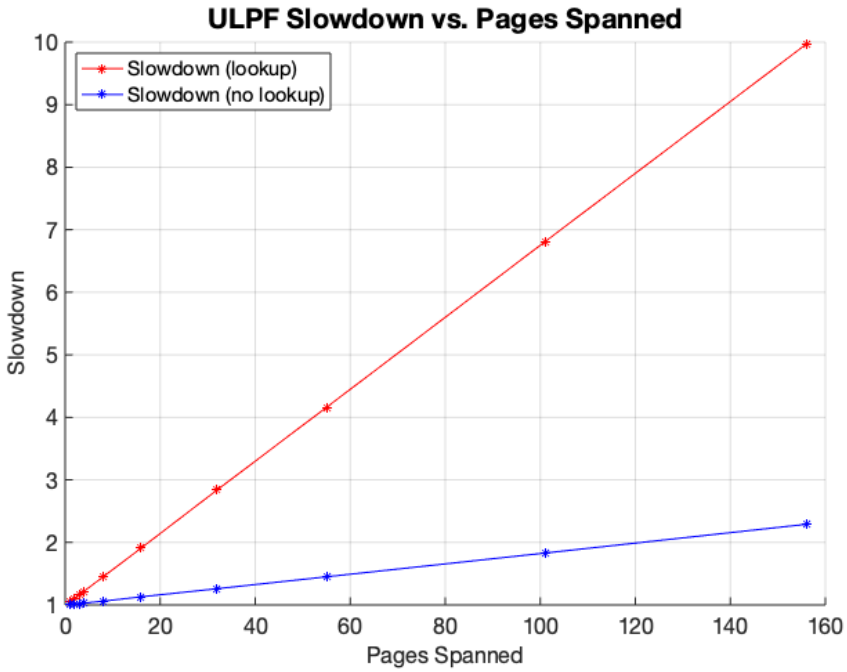

Figure 4.2: Slowdown of ULPF handlers vs. pages spanned by watchpoints

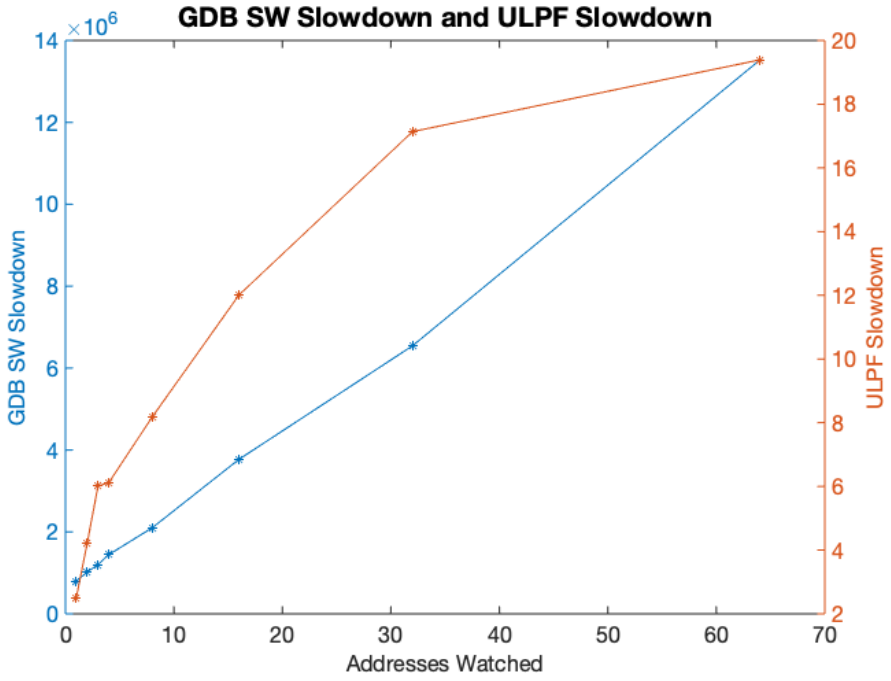

Figure 4.3: Slowdown of GDB software watchpoints vs ULPF

4.6 SPEC Benchmarks

We also tried to use our mechanism on applications in SPEC2006 benchmark suite. For each benchmark, we choose to watch all allocated heap memory. Due to several setbacks, we only successfully tested our mechanism on two benchmarks. We could not successfully run other programs within suites due to issues such as the disk image having insufficient memory and the kernel being outdated. We also had to disqualify programs that did not use heap memory; they accessed global memory. We chose to only watch heap memory because the program is explicit about when heap memory is allocated and freed.

Table 4.7: SPEC Watchpoints

Name	**WP Hit**	**FP**	**Tree Nodes**	**Slowdown (lookup)**	**Slowdown (no lookup)**
libquantum	7249859	348	95	6.8	1.8
lbm	4611909	3930	2	2.1	1.1

The many of the related work we mentioned previously did not run tests on the same set of SPEC benchmarks we did. EDDI [13] and MMP [12] ran on the SPEC 2000 benchmarks. MemTrace ran on the SPEC 2006 benchmarks and also benchmarked a watchpoint application. Their slowdown for the two programs we tested are shown in Table 4.8. Overall, our ULPF mechanism overhead is comparable to that of their binary translation implementation, shown by the overhead of ULPF without lookup. Since we must traverse a binary search tree every time we fault to differentiate between watched addresses and false positives, the overhead of ULPF watchpoints is significantly higher than that using MemTrace, as shown by the overhead of ULPF with lookup. However, it is important to note that ULPF applies to 64-bit applications, while MemTrace can only apply to 32-bit applications. While we have a higher overhead in runtime due to searching the watchlist tree with every fault, we allow the application to work with a larger address space because we do not use shadow memory. Also, to add watchpoints to a MemTrace program, the program must be recompiled, whereas recompiling is not necessary for a ULPF program.

Table 4.8: Comparison with MemTrace

Name	ULI **Slowdown (lookup)**	ULI **Slowdown (no lookup)**	**MemTrace**
libquantum	6.8	1.8	2.01
lbm	2.1	1.1	1.11

4.7 Summary

From our microbenchmark data, we found that there are two components that contribute to the overhead of running the program with addresses watched compared to having no addresses watched. The first is the number of pages spanned by the watched addresses. In other words,

this is the set of pages that the set of watchpoints are on. The second is the faulting cost. The faulting cost covers the cost of the faulting mechanism and the cost of running the handler. Our data shows that the slowdown of having watchpoints is linearly proportional to the product of the pages spanned and the faulting cost. However, the proportional constant is high, 4096, the number of addresses on a page.

The overhead of the ULPF faulting mechanism is low; there is around a 1% increase in overhead for every page watched. As we show in sparse and dense distribution of watched addresses, this overhead stays consistent as the number of pages increase. The size of the splay tree is constant through the execution of the program; all nodes are inserted into the tree before any memory is accessed. Therefore the cost of searching the splay tree is mostly the same from fault to fault. However, once we have watchpoints that are on different pages, the overhead starts increasing quickly. This is because for every address that is on a watched page, the faulting mechanism executes. This is the proportional constant mentioned earlier in our slowdown equation. However small the faulting cost is, having it execute 4096 times will contribute significantly to the runtime.

Overall, these results show that our hardware and software mechanisms for ULPFs are providing the low overheads that we hoped. Based on our current page-level faulting mechanism, there is little we can do to lower the proportional constant, which is the main contributor to the slowdown. Other than that, there might be optimizations we can make in the watchlist design for a shorter lookup, to minimize the handler cost.

Chapter 5

Improvements and Future Work

The ULPF design described in this thesis merely sets the foundation for future work; there are many issues that are not addressed. We have not discussed whether child processes will inherit their parents' user level handler or whether they will have the same permissions in their corresponding memory also set to read-only. We also have not deeply explored on whether giving users this functionality will affect program security.

There are several avenues for improving the performance of this mechanism that we did not have a chance to explore.

5.1 Watchlist Data Structure

5.1.1 Continue with Splay Trees

We selected splay trees as our watchlist structure based on our hypothesis that splay trees have advantages in terms of locality and being balanced were not supported across the board, as a unbalanced binary search tree showed less slowdown in some applications.

It is not clear whether using trees as our watchlist structure is the optimal choice. There needs to be a more extensive exploration of various implementations of BSTs (binary search trees) and balanced BSTs, e.g., red-black tree, AVL, splay tree, to see whether the one implementation has an improvement over others.

5.1.2 Bloom Filters

Instead of a tree structure that holds all watched addresses, there can be an additional level of indirection within the user library that uses bloom filters [2]. Bloom filters are probabilistic data structures that can determine non-membership with certainty. In the case of potential membership, a more concrete data structure, like a tree or hash-map, is needed. However, this method does not lend itself intuitively to represent ranges.

5.2 Optimizations

We decided to merge consecutive and overlapping address blocks to decrease the number of nodes in our watchlist tree. This proved to be useless when we were collecting data from microbenchmarks. Given that `malloc` will take some amount of memory as the header for a block, no consecutive allocated blocks will be merged. There is an argument to be made whether we should account for the header as part of the watched memory and thereby merging watched memory from two consecutive `malloc` calls. However, at this time, there is no method to determine whether a watched address is the return address of a `malloc` call, so we cannot make this assumption. We would need to make changes to the user library to make this distinction.

5.3 Future Exploration

Future work may include exploring other applications of a user-level faulting mechanism, like stack guarding. We have intentionally allotted two bits for every page in the user page tables for future applications.

Another interesting avenue to explore is having this user level faulting mechanism apply to only certain threads in multithreaded applications. This way, the user would detect whether some thread is writing into another thread's memory or changing some variable value. Unfortunately, existing hardware stores only states pertaining to individual processes rather than threads within processes. We have no way to differentiate between threads of the same process. For example, the MMU will refer only to %CR3, which holds the process page table, for address translation. Only the kernel has thread-level information, but requiring kernel interference will defeat the purpose of ULIs, which explicitly bypasses the kernel.

Chapter 6

Conclusion

This thesis shows that having a specialized faulting mechanism for user-installed software handlers is much more efficient than the current architecture of signal handling. There is a 100x speedup in using ULPFs for watchpoint implementation over using signal handlers. For implementing ULPFs, we essentially duplicated the kernel page tables and its corresponding page-walking logic within the MMU to store and lookup our user-defined permissions. We also added an additional path to be able to raise a ULPF through the TLB by caching user-level permissions as part of a TLB entry. For user space programs to be able to interact with these new paging structures, we installed new system calls.

We tested our mechanism on the watchpoints application, for which we developed a simple user library with an intuitive API to ease usability. In our microbenchmarks and SPEC benchmarks, we found advantages and disadvantages of our design. Our advantages are clear over existing signal handling methods. Even if we ignore our hardware capabilities of being able to "fault once" on an address without manipulating any permissions in memory, we save thousands of cycles by staying in the user stack and having a specialized mechanism. Compared to invoking the kernel page fault handler and passing execution to a user signal handler, our mechanism proves to be 100x faster. Compared to an application that does not use our mechanism, we only incur a 0.07% slowdown per page watched. Our disadvantages are that we are faulting on a page-level granularity. While the relationship of our overhead is linear with respect the pages spanned and the faulting cost, that the linear constant is 4096 causes our overhead to increase quickly as the number of watchpoints increase. In other words, our mechanism will fault 4096 times on a page with only a single watched address. To overcome this issue, we would need to reevaluate the existing memory management system, for the current system operates on the page level.

We have successfully shown through this design that it is possible to give users the power to define memory protection within their applications without compromising security of other processes. Our approach differs from many previous proposals in that we implement changes to the hardware, rather than limiting our changes to software components. Our mechanism is also more memory efficient compared to shadow memory implementations. Rather than allocating half, or an eighth of the entire memory space to store permissions, we allocate page tables with an need-by-need basis.

Chapter 7

Appendix

May it be a light to you in dark places,
when all other lights go out.

Galadriel
The Fellowship of the Ring, J.R.R. Tolkien

7.1 GDB Scripting

For the purpose of benchmarking, we needed such a script to automatically set watchpoints and run the program in its entirety to mimic the behavior of the corresponding ULPT program. GDB can be configured to run a script automatically upon startup. The contents of this script are typically found in `~/.gdbinit`. Each line is a command the user would have inputted into the GDB interface. With the script below, GDB will automatically load the program `gdb_test`, break at a certain line within the program, set a watchpoint at an address, and continue running until the program terminates. This eliminates any user interaction that GDB typically requires.

```
set pagination off            # Do not limit output length on screen
set logging off               # Turn off logging (can also be set to redirect to another file)
set can-use-hw-watchpoints 0  # Force GDB to use software watchpoints
file gdb_test                 # Name of file to run
break gdb_test.c:24           # Set breakpoint preemptively on line 24 of code file
run                           # Run the program to get variables in scope
watch *(block + 1234)         # block is base address of allocated memory. This sets a
                              # watchpoint on address (block + 1234)
continue                      # Continue with execution
```

Listing 7.1: Sample GDB Script

7.2 Making a New System Call

This is a quick overview of the files needed to be changed to install a new system call. All this information and more can be found on Google; I hope to make someone's life a bit easier.

My example will be using the Linux 4.19 kernel. More recent kernels will have a similar file structure, but the file or variable names might be a bit different. A grep over a classic system

call like `fork` or `wait` would list a superset of the files you would need to change. I will be using Ubuntu 18.04 as my distribution. Some Linux distros offer specialized packages to install a modified OS (like `dpkg` for Debian-based distros); it worked once and never again. I found the method I included below to be reliable and should be distro-insensitive, but I have only tried it on Ubuntu.

Download the kernel source code from the official website and untar it.

```
// Comment: I will be referring to your download directory as $OS_HOME

>> wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.19.191.tar.xz
>> tar -xzvf linux-4.19.191.tar.xz

// I will be referring to the linux base directory ($OS_HOME/linux-4.19.191) as $LINUX_HOME
```

The highest level file directory is split into key components of the kernel. I mostly focused in `mm` (memory management), `kernel`(most system calls), and a bit in `arch/x86/include/asm` (macros that bridge between the software and the hardware); I don't know much about the rest. While technically we can write system calls anywhere, I'll demonstrate this within the `$LINUX_HOME/kernel` subdirectory because that's where all the cool system calls are.

Within the `$LINUX_HOME/kernel` subdirectory:

```
// my_syscall.c : My first system call
// Super interesting and requires kernel privilege to execute
// Full path: $LINUX_HOME/kernel/my_syscall.c

asmlinkage long sys_mult_five(int input) {
  return 5 * input;
}
```

```
// Makefile
// Full path: $LINUX_HOME/kernel/Makefile

// This is super forgettable and frustrating to debug the resulting make errors.
obj-y = shed.o fork.o exec-domain.o ... \
      ... \
      my_syscall.o
```

Hopping over to `$LINUX_HOME/include`, we need to add your new function to the header files.

```
// syscalls.h
// Full path: $LINUX_HOME/include/linux/syscalls.h

// NOTE: If your syscall takes no parameters, you need to explicitly write void
asmlinkage long fork(void);
...
...
asmlinkage long sys_mult_five(int input);
```

```
// syscalls.h
// Full path: $LINUX_HOME/include/uapi/asm-generic/unistd.h

...
...
// Each syscall is registered with a trap gate, these gates are identified by IDs
#define __NR_statx 291
// This macro will generate and write the trap gate into the IDT for the corresponding ID and
    function
__SYSCALL(__NR_statx, sys_statx)
#define __NR_mult_five 292
__SYSCALL(__NR_mult_five, sys_mult_five);
```

This file defines all system calls and their respective call numbers if invoked from user space.

```
// syscall_64.tbl
// Full path: $LINUX_HOME/arch/x86/entry/syscalls/syscall_64.tbl

...
...
332 common statx sys_statx
// This number is important
333 common mult_five sys_mult_five
```

This is basically it. Time to build.

```
// In $LINUX_HOME, within the shell

// Compile everything
>> make -j $(nproc)

// Install kernel modules
>> sudo make modules_install

// Install the build images into your /boot directory
>> sudo make install

// For Debian-based distros, you don't have to update the GRUB config but if you want:
>> sudo update-initramfs -c -k 4.19.191
>> sudo update-grub
```

I personally found it useful for the boot process to stop at the GRUB menu instead of automatically booting into the first OS.

```
// Full path: /etc/default/grub
...
GRUB_TIMEOUT=-1
...
```

After this change, you must update grub.

```
>>  sudo update-grub
```

Great. Reboot and select your new kernel in the GRUB menu (you may have to look in Advanced options and select your kernel version).

```
>>  reboot
```

Now, let's invoke your new system call.

```
// User test for your new system call.
// Full path: $HOME/test_syscall.c

#include<stdio.h>

int main() {
  // syscall invokes system calls by their registered numbers in the syscall_64.tbl file
  int num = syscall(333, 8);
  printf("%d\n", num);
  return 0;
}

// Compile and run
>> ./test_syscall
40
```

Some things to know: These notes are not necessary if your system call does not involve any other process or memory components.

- The internal kernel memory allocator is `kmalloc`. The first argument is the same as `malloc`, the size of the memory you want to allocate. The second is flags for selecting a pool and/or specific action that you want to execute. For most purposes, this would be `GFP_KERNEL` if you want to allocate pages in kernel space (GFP stands for general free pages). The full documentation is online on the official kernel website.
- `task_struct` is the name of the general process control block. This holds all information (PID, process state, parent process, signal handlers, etc.) pertaining to a process. You can access the current running process by:

```
// task_struct is defined below
#include <linux/sched.h>
// get_current() is defined below
#include <asm/current.h>
void fun {
  struct task_struct* proc = get_current();
}
```

- Printing in kernel space uses `printk` with the same syntax as `printf` from `stdio.h`. You can access these logs using

```
>> dmesg
```

  in the shell.

7.3 Good References

- Linux source code explorer: Elixir Bootlin

  Most global variables, macros, and function declarations will hyperlink to a list containing their definition and usage files. You can find the same information through `grep`ping, but this website clearly distinguishes the definition files.

  Also contains LLVM, qemu, glibc, and several more repositories.
- Building concepts: Operating System Concepts by Abraham Silberschatz, Peter B. Galvin, Greg Gagne

  The classic dinosaur book – not exactly the most practical (look at entry below) but great to get to know the key kernel components.
- Kernel text reference: Understanding the Linux Kernel by Daniel P. Bovet, Marco Cesati

  This book covers, in extreme detail, Linux 2.6. Slightly outdated, but I found this book the perfect bridge between 15-410 and working on an actual kernel.
- Memory Management: Understanding the Virtual Memory Manager by Mel Gorman

  This text provides line-by-line explanations of the memory management system, including page faults, in Linux 2.6.
- The Intel 64 and IA-32 Architectures Software Developer Manuals

  Yes, they are massive. But if you want to play in their sandbox, these explain the rules (to a certain extent) really well.

Easter Eggs

- The Linux kernel usually goes by numbers for its release versions, but each release has a name. You can find them in the Makefile of the base directory. A personal favorite: 4.2.8 is "Hurr durr I'ma sheep."
- Developers (Linus) sometimes colorfully gripe about their struggles in the comments. A quick `grep` will find you some interesting bits.

  There seemed to be some dispute between Linus and Sun Microsystems. Prof. Eckhardt might know something about it.
- Due to a change in the Linux Code of Conduct, a patch has replaced all the f-words with "hug." Other profanities have not been adjusted.

Bibliography

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):17, August 2011. 4.3

[2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422426, July 1970. 5.1.2

[3] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, 2010. 2.2

[4] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. *SIGPLAN Not.*, 47(4):387400, March 2012. 2.2

[5] Chi keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *In Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005. 4.2

[6] Mathias Payer, Enrico Kravina, and Thomas R. Gross. Lightweight memory tracing. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 115–126, San Jose, CA, June 2013. USENIX Association. 4.2

[7] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, page 135148, USA, 2006. IEEE Computer Society. 4.2

[8] D. Stuart Ritchie and Gerald W. Neufeld. User Level IPC and Device Management in the Raven Kernel. In *Usenix Association Proc. Symp. on Microkernels and other Kernel Architectures*, pages 111–125, 1993. 2.2

[9] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012. 1.2

[10] Livio Soares and Michael Stumm. "flexSC: Flexible system call scheduling with exception-less system calls". In *OSDI*, volume 10, pages 1–8, 2010. 2.2

[11] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, page 256266, New York, NY, USA, 1992. Association for Computing Machinery. 3.1.1

[12] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 304316, New York, NY, USA, 2002. Association for Computing Machinery. 4.2, 4.6

[13] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC08/ETAPS08, page 147162, Berlin, Heidelberg, 2008. Springer-Verlag. 4.2, 4.6