The thesis of Ryan Pasculano was reviewed and approved by the following:

Trent Jaeger
Professor of Computer Science and Engineering
Thesis Co-Advisor

John Morgan Sampson
Associate Professor of Computer Science and Engineering
Thesis Co-Advisor

Chitaranjan Das
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering
Abstract

C and C++ have suffered from out-of-bounds errors since their inception. These languages are incapable of protecting from all out-of-bounds errors and there are no plans to add any kind of support to the language. Despite this major drawback, C and C++ projects continue to be maintained and actively developed. That leaves it up to hardware or operating system software to protect against this vulnerability. Many approaches have aimed to address this issue but they come with large overheads, radical hardware redesigns, or insufficient protections. Address Space Coloring aims to protect data from being reached by malicious out-of-bounds errors. This is achieved by modifying RISC-V and LLVM to partition load and store instructions into individual address spaces. By embedding these partitions into the machine instructions the address spaces become inaccessible from one another. Address Space Coloring is still in a developmental stage but looks to provide strong security from out-of-bounds errors with minor hardware changes.
# Table of Contents

List of Figures vi

List of Tables vii

Acknowledgments viii

Chapter 1
  Introduction 1

Chapter 2
  Background and Motivation 4
    2.1 Stack Overflow ................................. 4
    2.2 RISC-V ......................................... 5
    2.3 LLVM and Clang ............................... 6

Chapter 3
  Related Work 8
    3.1 Stack Protections ............................. 8
    3.2 Bounded pointers ............................. 10
    3.3 Tagged Architectures ......................... 11
    3.4 CHERI .......................................... 12
    3.5 Other Approaches ............................. 13

Chapter 4
  Design 14
    4.1 Restricted Load/Store ....................... 14
    4.2 Memory Space Layout .......................... 16
    4.3 Offset Analysis .............................. 17
    4.4 Simulation ................................... 18
    4.5 Compilation .................................. 18

Chapter 5
  Implementation 19
    5.1 RISC-V ISA .................................. 19
    5.2 Spike ........................................ 20
5.3 LLVM .............................................................. 21

Chapter 6
   Evaluation ......................................................... 22
   6.1 Offset Analysis ............................................. 22
   6.2 Approach .................................................... 23
   6.3 Initial Results .............................................. 24
   6.4 Future Evaluation .......................................... 26

Chapter 7
   Conclusions and Future Work ................................. 27

Appendix
   Raw Data from Offset Experiments ......................... 28

Bibliography ....................................................... 28
# List of Figures

2.1 RISCV instruction formats ........................................... 6

2.2 Diagram of Clang and LLVM compilation ............................ 7

3.1 Stack with canary and unsafe buffer ............................... 9

4.1 RISCV instruction formats ........................................... 15

4.2 New RISCV instruction formats ..................................... 15

4.3 Two potential coloring options for the address space ............. 16

4.4 Shifting window offset bits ......................................... 17

6.1 Static and dynamic instruction counts .............................. 23

6.2 Load and store by type ............................................. 23

6.3 Dynamic LSB counts ............................................... 23

6.4 Dynamic MSB counts ............................................... 23
List of Tables

5.1 Restricted instruction implementation ........................................... 20
1.1 Static and Dynamic Load and store counts ................................. 28
1.2 Load and store instructions as percentage of total instructions ........ 29
1.3 Breakdown of loads and stores by type ....................................... 29
1.4 Least significant bit ................................................................. 30
1.5 Most significant bit ................................................................. 30
Acknowledgments

Thanks to all my fellow researchers in the SIIS Lab at Pennsylvania State University for their contributions to this research. Special thanks to my advisors Trent Jaeger and John Morgan Sampson for their invaluable guidance.

This research was sponsored by the Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA) and National Science Foundation grant CNS-1801534. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not withstanding any copyright notation here on.

This material is based upon work supported by the National Science Foundation under Grant No. 1822923. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

To my parents, thank you for your support during my time in graduate school. To Melanie, thank you for helping me and always being there to encourage me to push myself.
Chapter 1  |  
Introduction

Buffer overflow attacks have been a major security vulnerability in C and C++ programs since their inception. These languages lack the bounds checking capabilities needed to prevent buffer overflows. Ultimately, it is up to the programmer to write code that is not vulnerable to buffer overflows. Due to the ongoing issue of buffer overflows, it seems that programmers will never adapt to writing code that is safe from buffer overflows. This has resulted in other approaches that attempt to fixing the issue using runtime software or hardware.

A buffer overflow is a type of spatial memory error. These errors exist in all programming languages but type safe languages such as Java use runtime checking to catch the errors and stop the program, usually throwing an out of bounds error. When a buffer overflow occurs in C or C++ there are no checks to catch it. Buffer overflows present a major vulnerability because it allows a malicious user to modify data that they shouldn’t have access to. With this ability the user can modify values such as a function’s return address resulting in altering the control flow of a program. Buffer overflows are the mechanism that allows for heap overflow, return-to-libc, and return oriented programming (ROP) attacks to be possible.

Many approaches have been taken to limit or solve spatial memory safety. Software approaches such as stack protections and bounded pointers have been proposed. Stack
canaries, a form of stack protections have been implemented into commercial operating systems. These software based solutions are either insufficient and do not protect against all types of attacks or they have very high runtime overheads resulting in their lack of adoption. Hardware solutions have been proposed in the form of Tagged Architectures and CHERI. These solutions are more thorough but require highly modified hardware resulting in their lack of adoption.

As we have seen software solutions have a high performance cost while hardware solutions require radical redesign. We can see from the lack of adoption that these approaches make too big of a sacrifice. A solution can not rely on a major change to the underlying hardware. Additionally, a software solution has not been found that provides guarantees of spatial memory safety while not having a large impact on performance. Address Space Coloring aims to find a middle ground, a hardware based solution with minimal architectural changes.

Address Space Coloring is a defense against buffer overflows that encodes security into RISC-V [1] instructions. Software solutions to buffer overflow result in insufficient protections. Other hardware solutions are costly due to their need to have a co-processor to handle security policy enforcement. Address Space Coloring introduces a new set of instructions to the compiler LLVM [2] and the RISC-V ISA. By embedding information into the instructions we can enforce policies at the lowest level.

Using the RISC-V ISA we have built and run tests to obtain an initial estimate on the additional instruction overhead. With these results we have modified the \texttt{imm} field to use reduced offsets. The modified LLVM compiler can be used to compile programs, the RISC-V toolchain can be used to link the programs to the necessary startup and tear down code, and the programs can be simulated using Spike [3]. The new instructions have been partially implemented into LLVM and Spike.

This paper’s contributions are as follows:
• Conducted analysis on the offset values of load and store instructions in RISC-V, based on static and dynamic tests, looking at load store width, most significant bit, and least significant bit.

• Modified the LLVM back-end for RISC-V to use reduced offsets to allow for embedding Address Space Coloring information into load and store instructions.

In this thesis we provide background and motivation for Address Space Coloring in Chapter 2. We discuss the relevant works in Chapter 3. The design of Address Space Coloring is explored in Chapter 4. Chapter 5 discusses implementation. Chapter 6 discusses the evaluation techniques, and in Chapter 7 we conclude.
Chapter 2  
Background and Motivation

C and C++ are popular programming languages and have large amounts of projects developed using them. These languages are highly capable and some of the fastest high level languages in use today. One of the many reasons these speeds are possible is the lack of bounds checking. Bounds checking ensures that when an array is accessed the access is within the allocated space of the array. In Java, for example if we try to execute the following snippet of code it would throw the error ArrayIndexOutOfBoundsException telling the programmer that they are trying to access an area of memory in an unintended way. When this happens it is called a buffer overflow.

```java
String str = "Hello World!"
char ch = str.charAt(20);
```

2.1 Stack Overflow

Stack overflow is a type of buffer overflow that occurs to a buffer located on the stack. This is the source of many attacks because the stack contains important information such as other local variables and the return address. If a buffer overflow is able to overwrite the return address then when it exits the function it will return to the overwritten address instead of the calling address. Write xor execute (W\X) is a security feature on
most operating systems that makes stack overflow attacks more complex but does not completely prevent them. Before W\&X was implemented, an adversary could encode executable code into a buffer and then overflow the buffer so the return address would point to the code they just inserted into the stack. These code injection attacks are no longer possible because now every segment of the address space is limited to either read or write permissions, but never both.

W\&X only prevents simple buffer overflow attacks. Other types of attacks are still possible including return-to-libc [4] and return oriented programming [5] (ROP) attacks. Both of these attacks are built out of gadgets. Gadgets are pieces of code that already exist in a program, end in a return statement, and complete some small task. ROP attacks occur when the return address of a function call is overwritten. The adversary can execute their own code by linking together a chain of gadgets to achieve more complex goals. A chain of gadgets can modify the stack, perform system calls, and much more. Return-to-libc is a similar attack that relies on overwriting the return address but uses the libc library as its code base to hijack. These attacks are easier to port between programs and systems because it is a consistent set of source code that exists in most programs.

2.2 RISC-V

Reduced Instruction Set Computer V (RISC-V) is open standard and therefore is free to use. The RISC-V ISA is built with extensions that allow for various features to be added based on the needs of the system. Some of the most common extensions include multiplication and division, atomic instructions, and floating point capabilities. These features make RISC-V a clean and easy to work with ISA. The layout of instructions are split into a few formats as shown in Figure 2.1.

The opcode is the first segment that is decoded and guides the compiler on how to
interpret the rest of the instructions. Funct3 holds information on which instruction type it is within the opcode category. Rd, rs1, and rs2 are the index of the registers to use for the instruction. The imm field is a constant that is encoded in the instruction. For load and store instructions the imm field allows for access to locations in memory that are close to addresses stored in a register without saving the exact address in an additional register. For example we can access multiple locations on the stack by modifying the imm field instead of moving the stack pointer or creating a temporary pointer. To make modifications to RISC-V we will be making use of Spike, a RISC-V ISA simulator [3], the RISC-V gnu toolchain [6], and the RISC-V proxy kernel [7].

2.3 LLVM and Clang

LLVM and Clang work together as a compiler tool similar to gcc. Clang is a front end that will compile code into the LLVM intermediate representations (IR). The LLVM IR is a high level assembly language. Users can use the IR to create compiler passes or insert additional code. LLVM is the backend that is able to compile to many different machines including RISC-V. A high level overview of LLVM and clangs compilation process can be seen in Figure 2.2. Another benefit of LLVM is that users are able to break down the compilation process into individual passes that are written in C++ so the user is able to
modify the passes to suit their needs. This will be helpful for adding new instructions to RISC-V. The RISC-V backend for LLVM was introduced in version 10.0.0 so we will need to a minimum of version 10.0.0.
Chapter 3  |  Related Work

Many techniques have been introduced that provide protections for the address space. Some of these techniques provide protections for only the stack while others protect both the stack and the heap. In this chapter, we will explore a few of those techniques.

3.1 Stack Protections

Stack protection techniques include Stack Canaries, Shadow Stack, and Safestack. Stack canaries [8,9] work to protect the return address from buffer overflow. When a buffer on the stack overflows, it has the potential to overwrite other data on the stack. The stack canary is a randomly selected integer that is placed directly below the return address. Before the return address is used the canary is checked and if it has changed value it will halt the program before the adversary can take control. If an adversary were to overflow a buffer and overwrite the return address, they would have to first overwrite the canary.

Stack canaries do have multiple shortcomings. They do not protect all data on the stack. We can see in 3.1 if the buffer were to overflow then the local variable could still be overwritten. Another weakness of the canary is that if the canary value can be leaked, then the value can be overwritten with the same value and the canary check will not detect any tampering.

Shadow Stack [10,11] provides more guarantees over a stack canary. Shadow stacks
use a secondary stack to store information to ensure that it will not be overwritten by unsafe buffers. When a function is called, it will write the return address in both the stack and the shadow stack. If a buffer overflow does occur during the execution of the function then the return address on the stack will be overwritten. When the function returns, it will compare the return address values on the stack and shadow stack, similar to how the canary value is checked against a known value. If the two return addresses are different then the program will halt. This is an improvement over stack canaries but it still does not protect against tampering with other data on the stack.

A third type of stack protection mechanism is Safe stack [12]. Safe stack uses two stacks similar to shadow stack. In safe stack, the compiler will determine which elements are not exploitable and will place them in the safe stack. The exploitable elements will be left on the stack. This technique requires hardware to make a safe stack that can only be accessed with escalated permissions or instructions. Without hardware support this technique can not be enforced. This technique like all stack protection techniques does not protect the heap from buffer overflows.
3.2 Bounded pointers

The technique of using bounded pointers to enforce spatial memory safety has a variety of approaches. Bounded pointers have associated metadata stored to enforce spatial safety. The metadata usually includes two pieces of information, a base address and either a size or an end address. This metadata allows for additional checks to ensure that a pointer does not extend beyond its valid range.

Safe pointers [13] adapt pointers into a data structure that includes the pointer, base, size, and a few other pieces of metadata. By creating wrapper malloc and free functions, the programmer can interact with the new pointer types without having to make major changes to how they program. With few exceptions safe pointers provide full spatial memory safety but have runtime overheads of 130% to 540%.

Hardbound [14] is a hardware approach to bounded pointers. The metadata is stored in a different area of the address space and special hardware mechanisms deal with checking for spatial memory safety when the pointers are created, used, and destroyed. Hardbound’s implementation requires an additional L1 tag cache and the added metadata needs to be accessed on every load and store instruction. While the runtime overheads are around 10% there are major changes needed for the hardware and the space and energy requirements are unknown.

Softbound [15] uses a software only approach but still conceals the implementation inside the compiler to encourage adoption among programmers. A separate metadata facility stores base and bound information. On load and store instructions, additional code is inserted at compile time to check for spatial memory safety. With a separate metadata facility, uninstrumented code will be able to execute alongside the instrumented code because the pointers representation is unchanged. Softbound considers using a store-only model that only checks for memory safety on store instructions, but does not
claim that it will be fully secure. With full checking the runtime overheads become 79% on average.

Baggy bounds and low fat pointers [16,17] use a more efficient technique or storing metadata within the pointer. Of the 64-bits in modern systems not every bit is used. With these spare bits, low fat pointers indexes a table based on the value of the spare bits. This table is then able to quickly check if any actions made by the pointer are safe. Baggy bounds uses the spare bits to provide an approximate size for the pointer. To ensure spatial memory safety, the allocator needs to ensure that allocation blocks are rounded up to the nearest power of two. This is because with the limited spare bits, the precision of the bound information is not as precise as other methods. The loose spacing of memory is a drawback to not having to store metadata in a separate location. Additionally, there are many edge cases that these methods do not cover including sub object bounds protection, and casting integers to pointers.

### 3.3 Tagged Architectures

Another technique that provides a wide range of security guarantees is tagged architectures [18–21]. These techniques rely on treating each processor instruction as a function of the form:

\[ F(\text{PC}_{in}, I, R_1, R_2, M_{in}) \rightarrow (\text{PC}_{out}, M_{out}) \]

\( \text{PC}_{in} \) and \( \text{PC}_{out} \) represent the Program Counter for the instruction and the next instruction. \( I \) represents the instruction at the address of PC. \( R_1 \) and \( R_2 \) are the registers that are accessed by the specific instructions. \( M_{in} \) and \( M_{out} \) represent the memory states before and after the instruction is executed. Every time an instruction is executed an additional component of the processor will validate that the function input maps to a valid output. If it detects an invalid function than the process will halt while an external
mechanism will determine whether the program flow is new and add it to the cache of valid flows or if it is malicious.

To implement return address stack protections like in shadow stack or stack canary, we can enforce checks on all instructions that read or write return addresses on the stack. This enforces that only instructions that are allowed to write to the segments of memory that contain the return address are the instructions that are intended. If another instruction tries to overwrite the return address, the tagging mechanism will detect an invalid flow and halt the program.

Tagged architectures are also capable of more complex security mechanisms like taint tracking. To implement taint tracking on a tagged architecture new flows need to be added dynamically as data moves through a program. Many other security properties can be enforced using a tagged architecture.

These security properties do come with drawbacks. In order to implement any of these properties, major changes need to be made at the hardware level. Changes need to be made to the processor to allow for fast checking of valid flows. The cache also needs modifications to make space to store the list of valid flows. This additional cache puts more strain on the cache and takes away from the instruction and data caches. The high hardware costs makes the adoption of tagged architectures unlikely.

### 3.4 CHERI

The Capability Enhanced Hardware RISC Instructions (CHERI) ISA is another approach to spatial memory safety. [22] The CHERI project is being developed by SRI International and University of Cambridge. Their approach to memory safety is through hardware enhancements and custom MIPS instructions. Pointers are implemented as capabilities instead of virtual addresses. Capabilities contain the pointer, bound metadata, and permissions. To implement these capabilities, a 128-bit pointer is created with a 64-bit
pointer and a 64-bit tag. Special registers and hardware are needed to fully realize the
capabilities. CHERI has only been implemented on a FPGA and in simulations. CHERI
can not run on a CPU because of the major hardware modifications that are needed.

3.5 Other Approaches

Other approaches have been taken but are less popular. Nile [23] uses a programmable
coprocessor to protect the stack by creating a shadow stack on the coprocessor and
comparing any return addresses. Intel introduced their Memory Protection Extension
(MPX) [24] as a hardware feature to allow for more secure pointer use. MPX has been
used in papers [25] for memory protection, but it has not seen much adoption. Operating
systems and compilers have started to remove the use of MPX due to its lack of benefits.
Lastly, Address Space Layout Randomization (ASLR) is a technique to obfuscate the
location of data in the address space. This technique is widely adopted in Operating
systems due to its minimal overhead. ASLR does not prevent any attacks but it does make
any attack more difficult to perform. Cryptographic Control Flow Integrity (CCFI) [26]
attempts to enforce CFI by adding message authentication codes (MACS) to control
flow elements. CCFI is still vulnerable to replay attacks, using old codes because of the
inability to select a random nonce value.
Chapter 4  
Design

In this chapter we will discuss the design elements that need to be considered in order to implement address space coloring. The design considerations covered are instruction design (Section 4.1), memory space layout (Section 4.2), number of color bits (Section 4.3), and simulation and compilation requirements (Sections 4.4 and 4.5)

4.1 Restricted Load/Store

RISC-V instructions follow a standardized layout in order to remain understandable and assist in decoding. The RISC-V load instructions use the format type I and the store instructions use the format type S. Load instructions have the fields of imm, rs1, funct3, rd, and opcode. Store instructions are similar but split the imm field into two sections and have an rs2 field instead of an rd field. We can see the format for these instruction types in Figure 4.1. In order to encode extra information into these custom instructions we need to create space within the instructions. The opcode, funct3, and register fields can not be shrunk or removed without losing functionality of the instruction. This leaves the imm field. With 12 bits in the imm field we can remove all of the bits and force the restricted loads and stores to have an offset of zero. Another option is to remove a subset of bits from the imm field and have fewer bits allocated to coloring the address space. We consider these two options in Section 4.3. We can see what these new instruction
CORE INSTRUCTION FORMATS

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>Opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>Opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>imm[31:12]</td>
<td>rd</td>
<td>Opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.1. RISCV instruction formats

CORE INSTRUCTION FORMATS

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>Opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>color</td>
<td>imm[4:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>Opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>color</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>imm[4:0]</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>imm[31:12]</td>
<td>rd</td>
<td>Opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.2. New RISCV instruction formats

layouts will look like in Figure 4.2.

In order to build these custom instructions we can keep the same naming conventions used for the funct3 in the regular load and store instructions. We have to find an unused opcode because our instructions will have a new format. The rd, rs1, and rs2 fields will remain unchanged. The imm field will be split into two sections to allow for color to be embedded into the instruction. In order to use these bits for assigning color they need to be implemented into the execution of the instruction. This can be done by concatenating the color bits onto the access to memory. We will discuss the effects this has on memory space layout in the next section.
Figure 4.3. Two potential coloring options for the address space

4.2 Memory Space Layout

By concatenating the color bits to a reference to memory we will create a new memory space address that can only be accessed by load and store instructions which carry the same color. This will also partition the address space into as many parts as there are colors. The location of the color bits will shape how memory appears and how it is interacted with by the processor and cache. If we place the color bits near the most significant bit (MSB) it will create large chunks of memory of the same color clumped together. If we place the color bits near the least significant bits (LSB) it will make many small partitions of each color. Figure 4.3 shows what these two layouts would look like with MSB coloring on the left and LSB on the right. These two approaches will have effects on the cache that will not be fully understood until implemented and tested. Small partitions mean that different color memory can reside in the same page or even the same cache block. This could be good but if some colors are unused then they will be empty space taking up part of the cache or page. Large blocks mean frequent use of a
single color will be fast because of its proximity, but usage of many colors will result in many different pages and cache lines needing to be loaded.

### 4.3 Offset Analysis

Some testing and analysis was conducted in order to determine the additional instruction overhead incurred for converting the entire imm field to the color field. A variety of factors were examined including static and dynamic instruction counts, frequency of load and store instructions by type, most significant non zero bit, and least significant non zero bit. The data from these findings are in Section 6.1.

The optimal solution we found was to use a shifting window for the imm field as shown in Figure 4.4. We can start by splitting the imm field into 6 bits of color and 6 bits of imm. These 6 bits will be different for each size load and store instruction. The byte size will be the 6 least significant bits but the double size will be bits 3 to 8. A full immediate field allows for offsets that range from -2048 to 2047. We found that most programs have less than 10% negative instructions so we removed the ability to have negative offsets. With this implementation byte offsets range from 0 to 63, and double offsets range from 0 to 504 in increments of 8. By our analysis we can estimate that this will result in an average of 20% of instructions having too large of an offset or a negative offset that the new custom instructions will not be able to handle.
4.4 Simulation

In order to simulate these new instructions there will be a number of steps. The calculations made for offsets will need to be adapted due to the change in the offset field. We will need to add logic to properly detect these new instructions as well as execute them properly. A major change to the simulator will be the additional logic needed to incorporate the color into load and store actions. To minimize runtime we should look for ways to optimize cache usage within the simulator. These are the main considerations when adapting the simulator to work with the modifications.

4.5 Compilation

To compile these new instructions we will need to make modifications to LLVM, the backend of the compiler. The process of converting IR to Machine code takes the following steps: IR → Selection DAG → Machine DAG → Machine IR → Machine Instructions. We will need to analyze the compilation process to find the appropriate location to make our modifications and then ensure that no other passes interfere with the modifications later in the process. The changes that will need to be made are limiting the offset sizes and adding consistent coloring logic. The more unique colors that the compiler makes use of the more memory safety it will be able to provide.
Chapter 5 | Implementation

With the design features laid out we can begin to implement Address Space Coloring using RISC-V, Spike, and LLVM. The next sections will discuss the steps taken to add Address Space Coloring as well as some additional work that is planned.

5.1 RISC-V ISA

To implement the instructions into the RISC-V language we need to clearly define the instructions. We have discussed earlier that we will keep the funct3 section consistent with the original load and store instructions. A new opcode is needed because of the new instruction format and the lack of room within the existing load and store opcodes. For this we selected opcode 0x0b for loads and 0x2b for stores. This is also a convenient pairing because the traditional load and store opcodes are 0x03 and 0x23 respectively. This keeps the two instruction types seperated by a factor of 0x20. A logical mnemonic for the instructions is to just add an 'r' to the beginning of each instruction. For a restricted load word we use the mnemonic rlw. Another benefit of this organization is that we can use the funct3 values to also encode the shifting value of the immediate for our shifting window.

The execution of the new instructions is implemented using Verilog. The Verilog code used is displayed in the right most column of Table 5.1. Let’s examine the rld instruction.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Opcode</th>
<th>funct3</th>
<th>Verilog Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restricted Load Byte</td>
<td>rlb</td>
<td>0x0b</td>
<td>0x0</td>
<td>( R[rd] = {56'bM[7], M[1'b1, color, R[rs1]+imm&lt;&lt;funct3(50:0)]} (7:0) )</td>
</tr>
<tr>
<td>Restricted Load Half</td>
<td>rlh</td>
<td>0x0b</td>
<td>0x1</td>
<td>( R[rd] = {56'bM[(15), M[1'b1, color, R[rs1]+imm&lt;&lt;funct3(50:0)]} (15:0) )</td>
</tr>
<tr>
<td>Restricted Load Word</td>
<td>rlw</td>
<td>0x0b</td>
<td>0x2</td>
<td>( R[rd] = {56'bM[(31), M[1'b1, color, R[rs1]+imm&lt;&lt;funct3(50:0)]} (31:0) )</td>
</tr>
<tr>
<td>Restricted Load Double</td>
<td>rld</td>
<td>0x0b</td>
<td>0x3</td>
<td>( R[rd] = M<a href="">1'b1, color, R[rs1]+imm&lt;&lt;funct3(50:0)</a> )</td>
</tr>
<tr>
<td>Restricted Store Byte</td>
<td>rsb</td>
<td>0x0b</td>
<td>0x0</td>
<td>( M<a href="">1'b1, color, R[rs1]+imm&lt;&lt;funct3(50:0)</a> = R<a href="">rs2</a> )</td>
</tr>
<tr>
<td>Restricted Store Half</td>
<td>rsh</td>
<td>0x0b</td>
<td>0x1</td>
<td>( M<a href="">1'b1, color, R[rs1]+imm&lt;&lt;funct3(50:0)</a> = R<a href="">rs2</a> )</td>
</tr>
<tr>
<td>Restricted Store Word</td>
<td>rsw</td>
<td>0x0b</td>
<td>0x2</td>
<td>( M<a href="">1'b1, color, R[rs1]+imm&lt;&lt;funct3(50:0)</a> = R<a href="">rs2</a> )</td>
</tr>
<tr>
<td>Restricted Store Double</td>
<td>rsd</td>
<td>0x0b</td>
<td>0x3</td>
<td>( M<a href="">1'b1, color, R[rs1]+imm&lt;&lt;funct3(50:0)</a> = R<a href="">rs2</a> )</td>
</tr>
</tbody>
</table>

Table 5.1. Restricted instruction implementation

We are loading a value into the register \( r_d \) which is expressed as \( R[r_d] \). This value comes from memory, \( M \), but we need to construct the address in memory first. This is done by calculating the offset and concatenating it with the color bits. To calculate the offset we shift the \( \text{imm} \) field by the value of \( \text{funct3} \) and add it to the base value stored at \( r_s_1 \). The other instructions have similar implementations but only make use of a portion of the 64 bit width. The additional complexity of the other instructions is from selecting a subset of the bits.

5.2 Spike

To add these instructions to the toolchain, the opcodes we created must be added to the system. Once that is done the \textit{opcode} program can generate the necessary matching and masking values. These match and mask values are added to the list of all match and mask values for all instructions which help with instruction decoding. The behaviour of each instruction is implemented in a \texttt{.h} file of the mnemonics name. This will allow the
simulator to execute the instructions when they appear in the simulator. The behaviour of the instructions has not been implemented into the simulator yet. The compilation section has not been completed so the simulator can not be properly tested at this point. The .h files will have an implementation that is similar to the last column of Table 5.1.

5.3 LLVM

For compiling the programs to use the new instructions the first step was to identify which pass would allow for the most seamless integration of the custom instructions. The compiler has over 30 passes between the IR and the machine instructions. These instructions will be modified versions of the original load and store instructions. This means that waiting until the load and store instructions have been created and as the offsets are being finalized would be the best time to add the new instructions. Making changes earlier would not work because the offsets are not yet known. Doing it later would mean that we would lose access to information that will help determine color of each load and store. By compiling the programs and printing the code after each pass we were able to find that within the prologue and epilogue insertion pass, the offsets were finalized. This pass also included code to check the offsets and generate an additional instruction if the offset was too large for a regular load or store. We were able to modify this to fit our needs.

The next step of the process is to add logic to select colors and encode the colors into the RISC-V instructions. This work has not been completed yet.
Chapter 6 | Evaluation

This section will discuss the evaluation of our intermediate results and then go on to discuss the testing and evaluation plans for the completed implementation.

6.1 Offset Analysis

Based on test results we can use the bounds of 30% and 40% as the lower and upper bound for the number of instructions that are loads and stores. If we were to remove the imm field entirely it would result in an additional instruction for every load or store instruction giving an additional 30% to 40% increase in dynamic instructions. This is a large cost so we should try and keep part of the imm field to minimize the additional instructions. From Figure 6.1 we can see there are a large group of instructions with an offset of zero and also few instructions using the largest bits. From Figure 6.2 we can see that for most programs over 90% of loads are of the word or double size. This is important because that means that they are the types of load instructions that should be prioritized. Figure 6.3 shows the least significant non zero bit. Further analysis showed that the half size never uses the least significant bit, the word size never uses the two least significant bits, and the double size never uses the three least significant bits. This means that all half instructions are aligned on half word sizes, word sized instructions are word aligned, and double size instructions are aligned on every other word. From
Figure 6.1. Static and dynamic instruction counts

Figure 6.2. Load and store by type

Figure 6.3. Dynamic LSB counts

Figure 6.4. Dynamic MSB counts

Figure 6.4 we can see that the most significant bit drops starting at bit 5 and stays low from bit 8 to 11. With these findings we can create a solution that allows for a minimal need to rewrite instructions due to large offsets.

This will result in an additional 6% to 8% increase in instruction size while allowing for 128 unique colors. These are reasonable runtime costs but we will revisit the trade off between the number of bits in the imm and color fields once we have initial data.

6.2 Approach

During initial testing of the LLVM compiler for the RISC-V backend, we discovered that the code generated does not include the necessary initialization and teardown code needed to run on Spike. To avoid this issue, the programs we want to run need to be compiled to object files using Clang. We can then write a wrapper program that calls our test program. The wrapper program can be compiled to an object file using riscv-gcc,
which is included in the RISC-V GNU Compiler Toolchain. The two object files can be linked using the linker in the same toolchain and then run on Spike. Currently, we are working on a way to generate instruction execution counts for only the code executed from the linked object file.

### 6.3 Initial Results

An initial test using the function add_one resulted in an increase from 12 RISC-V instructions to 16. These 4 additional instructions came from accessing arguments on the stack that had a negative offset, but with optimizations could be reduced to a single additional instruction. The main optimization relies on spatial locality. When a negative or large offset is used, it is often not used in isolation. By identifying multiple instructions that are negative we can use one additional addi instruction to create a temporary pointer for multiple loads and stores instead of calculating a new value each time. This initial test was static but the function does not have any branches so a dynamic test will produce the same results. Below is the RISC-V code for the Regular LLVM compiler and then the modified version. We can see that the current implementation uses an li and add instruction to access the offset of -20 but a single addi instruction could be used. This modification would cut the additional instructions needed in half. Additionally, the same calculation of -20 is performed twice, (105de and 105e4) by knowing that the value has already been calculated, an additional instruction could be optimized out.

**Listing 6.1.** Unmodified add_one in RISC-V

00000000000105d4 <add_one>:

- 105d4: 1101 addi sp , sp, -32
- 105d6: ec06 sd ra , 24(sp)
- 105d8: e822 sd s0 , 16(sp)
Listing 6.2. Modified add_one in RISC-V

00000000000105d4 <add_one>:

105d4: 1101 addi sp ,sp,−32
105d6: ec06 sd ra ,24(sp)
105d8: e822 sd s0 ,16(sp)
105da: 1000 addi s0 ,sp ,32
105dc: 85aa mv a1 ,a0
105de: 55b1 li a1 ,−20
105e0: 95a2 add a1 ,a1 ,s0
105e2: c188 sw a0 ,0(a1)
105e4: 5531 li a0 ,−20
105e6: 9522 add a0 ,a0 ,s0
105e8: 4108 lw a0 ,0(a0)
105ea: 2505 addi a0 ,a0 ,1
105ec: 6442 ld s0 ,16(sp)
105ee: 60e2 ld ra ,24(sp)
105f0: 6105 addi sp ,sp ,32
Due to the issues with compiling and executing programs, we are limited in our ability to test. We can not use any of the usual benchmarks like SPEC. For evaluating Address Space Coloring we will need to make our own test suite. This has not been completed as work is still being done on implementing the compiler.

6.4 Future Evaluation

In addition to creating the test suite, there is also the work of conducting testing. We plan on running a number of tests to establish the cost incurred by the modifications to RISC-V. We plan on comparing the runtime, instruction execution count, and code size of the modified code compared to the unmodified LLVM compiler. In addition we will also estimate the space and energy overheads once we have finalized a design.
Address Space Coloring is a modified RISC-V ISA that partitions memory to protect against spatial memory errors. The additional instruction overhead is estimated to be below 8% and runtimes should also remain small. Address Space Coloring requires small changes in hardware compared to tagged architectures and CHERI. The design process has been completed for Address Space Coloring and the implementation and evaluation has been planned but has not yet finished.

Through testing we will be able to refine our design to optimize for efficiency. Varying the number of bits designated for color will change the additional instruction overhead. Reintroducing negative offsets would also have a similar effect. By shifting the location of the color bits in the memory address we can create more small partitions or fewer large partitions. These changes will have an effect on the cache due to the sparseness it will create in memory. Looking into these effects will help refine the design. There may also be ways to compress this sparseness in memory but they will not be able to be fully realized until a prototype is built. Address Space Coloring is a promising way to embed security hardening into computers at the machine instruction level.
### Appendix

#### Raw Data from Offset Experiments

<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>Dynamic</th>
<th>Coverage</th>
<th>Change in code size</th>
<th>Change in Instr count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr</td>
<td>17539</td>
<td>1494</td>
<td>8.52%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L/S</td>
<td>4368</td>
<td>669</td>
<td>15.32%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neg</td>
<td>162</td>
<td>14</td>
<td>8.64%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 Bits</td>
<td>602</td>
<td>68</td>
<td>10.16%</td>
<td>121.47%</td>
<td>124.82%</td>
</tr>
<tr>
<td>1 Bits</td>
<td>2</td>
<td>3</td>
<td>10.61%</td>
<td>121.46%</td>
<td>124.79%</td>
</tr>
<tr>
<td>2 Bits</td>
<td>28</td>
<td>7</td>
<td>11.66%</td>
<td>121.30%</td>
<td>124.59%</td>
</tr>
<tr>
<td>3 Bits</td>
<td>19</td>
<td>13</td>
<td>13.60%</td>
<td>121.19%</td>
<td>124.41%</td>
</tr>
<tr>
<td>4 Bits</td>
<td>579</td>
<td>115</td>
<td>30.79%</td>
<td>117.89%</td>
<td>120.45%</td>
</tr>
<tr>
<td>5 Bits</td>
<td>864</td>
<td>173</td>
<td>56.65%</td>
<td>112.97%</td>
<td>114.54%</td>
</tr>
<tr>
<td>6 Bits</td>
<td>607</td>
<td>95</td>
<td>70.85%</td>
<td>109.50%</td>
<td>110.54%</td>
</tr>
<tr>
<td>7 Bits</td>
<td>580</td>
<td>77</td>
<td>82.36%</td>
<td>106.20%</td>
<td>106.79%</td>
</tr>
<tr>
<td>8 Bits</td>
<td>281</td>
<td>25</td>
<td>86.10%</td>
<td>104.60%</td>
<td>105.05%</td>
</tr>
<tr>
<td>9 Bits</td>
<td>448</td>
<td>14</td>
<td>88.19%</td>
<td>102.04%</td>
<td>102.41%</td>
</tr>
<tr>
<td>10 Bits</td>
<td>86</td>
<td>27</td>
<td>92.23%</td>
<td>101.55%</td>
<td>103.48%</td>
</tr>
<tr>
<td>11 Bits</td>
<td>110</td>
<td>38</td>
<td>97.91%</td>
<td>100.92%</td>
<td>100.94%</td>
</tr>
</tbody>
</table>

*Table 1. Static and Dynamic Load and store counts*
### Table 2. Load and store instructions as percentage of total instructions

<table>
<thead>
<tr>
<th>Type</th>
<th>myprog</th>
<th>cmds</th>
<th>cmdreverse</th>
<th>lab0</th>
<th>student</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr</td>
<td>46656</td>
<td>36195</td>
<td>1,524</td>
<td>52050</td>
<td>1523</td>
</tr>
<tr>
<td>L/S</td>
<td>19164</td>
<td>7313</td>
<td>723</td>
<td>20288</td>
<td>652</td>
</tr>
<tr>
<td>Neg</td>
<td>946</td>
<td>2547</td>
<td>41</td>
<td>2287</td>
<td>13</td>
</tr>
<tr>
<td>0 Bits</td>
<td>1911</td>
<td>1246</td>
<td>82</td>
<td>2,111</td>
<td>91</td>
</tr>
<tr>
<td>1 Bits</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2 Bits</td>
<td>7</td>
<td>1098</td>
<td>9</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>3 Bits</td>
<td>13</td>
<td>61</td>
<td>17</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>4 Bits</td>
<td>2435</td>
<td>407</td>
<td>127</td>
<td>2,776</td>
<td>113</td>
</tr>
<tr>
<td>5 Bits</td>
<td>4216</td>
<td>701</td>
<td>212</td>
<td>4,607</td>
<td>179</td>
</tr>
<tr>
<td>6 Bits</td>
<td>3646</td>
<td>443</td>
<td>102</td>
<td>3,223</td>
<td>93</td>
</tr>
<tr>
<td>7 Bits</td>
<td>2727</td>
<td>376</td>
<td>66</td>
<td>2,278</td>
<td>74</td>
</tr>
<tr>
<td>8 Bits</td>
<td>580</td>
<td>211</td>
<td>19</td>
<td>404</td>
<td>19</td>
</tr>
<tr>
<td>9 Bits</td>
<td>1031</td>
<td>92</td>
<td>4</td>
<td>823</td>
<td>4</td>
</tr>
<tr>
<td>10 Bits</td>
<td>1313</td>
<td>79</td>
<td>1</td>
<td>1,248</td>
<td>1</td>
</tr>
<tr>
<td>11 Bits</td>
<td>336</td>
<td>45</td>
<td>39</td>
<td>508</td>
<td>35</td>
</tr>
<tr>
<td>Percent</td>
<td>41.08%</td>
<td>20.20%</td>
<td>47.44%</td>
<td>38.98%</td>
<td>42.81%</td>
</tr>
</tbody>
</table>

### Table 3. Breakdown of loads and stores by type

<table>
<thead>
<tr>
<th>Type</th>
<th>myprog</th>
<th>cmds</th>
<th>cmdreverse</th>
<th>lab0</th>
<th>student</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>821</td>
<td>117</td>
<td>40</td>
<td>1433</td>
<td>35</td>
</tr>
<tr>
<td>Half</td>
<td>314</td>
<td>4811</td>
<td>38</td>
<td>459</td>
<td>36</td>
</tr>
<tr>
<td>Word</td>
<td>1878</td>
<td>211</td>
<td>133</td>
<td>1612</td>
<td>71</td>
</tr>
<tr>
<td>Double</td>
<td>16151</td>
<td>2174</td>
<td>512</td>
<td>16784</td>
<td>510</td>
</tr>
<tr>
<td>Quad</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>19164</td>
<td>7313</td>
<td>723</td>
<td>20288</td>
<td>652</td>
</tr>
<tr>
<td></td>
<td>cmdreverse</td>
<td>cmds</td>
<td>lab</td>
<td>myprog</td>
<td>student</td>
</tr>
<tr>
<td>-------</td>
<td>------------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>Instr</td>
<td>1,524</td>
<td>36195</td>
<td>52050</td>
<td>46656</td>
<td>1523</td>
</tr>
<tr>
<td>L/S</td>
<td>723</td>
<td>7313</td>
<td>20288</td>
<td>19164</td>
<td>652</td>
</tr>
<tr>
<td>Neg</td>
<td>41</td>
<td>2547</td>
<td>2287</td>
<td>946</td>
<td>13</td>
</tr>
<tr>
<td>0 Bits</td>
<td>16</td>
<td>37</td>
<td>98</td>
<td>174</td>
<td>16</td>
</tr>
<tr>
<td>1 Bits</td>
<td>19</td>
<td>1174</td>
<td>89</td>
<td>60</td>
<td>19</td>
</tr>
<tr>
<td>2 Bits</td>
<td>37</td>
<td>115</td>
<td>641</td>
<td>622</td>
<td>22</td>
</tr>
<tr>
<td>3 Bits</td>
<td>291</td>
<td>1334</td>
<td>8,258</td>
<td>8,157</td>
<td>285</td>
</tr>
<tr>
<td>4 Bits</td>
<td>177</td>
<td>545</td>
<td>4,248</td>
<td>4,268</td>
<td>151</td>
</tr>
<tr>
<td>5 Bits</td>
<td>45</td>
<td>193</td>
<td>1,806</td>
<td>2,001</td>
<td>37</td>
</tr>
<tr>
<td>6 Bits</td>
<td>11</td>
<td>101</td>
<td>661</td>
<td>920</td>
<td>14</td>
</tr>
<tr>
<td>7 Bits</td>
<td>4</td>
<td>16</td>
<td>46</td>
<td>54</td>
<td>4</td>
</tr>
<tr>
<td>8 Bits</td>
<td>0</td>
<td>5</td>
<td>43</td>
<td>51</td>
<td>0</td>
</tr>
<tr>
<td>9 Bits</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10 Bits</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11 Bits</td>
<td>82</td>
<td>1246</td>
<td>2,111</td>
<td>1,911</td>
<td>91</td>
</tr>
</tbody>
</table>

**Table 4.** Least significant bit

<table>
<thead>
<tr>
<th></th>
<th>myprog</th>
<th>cmds</th>
<th>cmdreverse</th>
<th>lab0</th>
<th>student</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr</td>
<td>46656</td>
<td>36195</td>
<td>1,524</td>
<td>52050</td>
<td>1523</td>
</tr>
<tr>
<td>L/S</td>
<td>19164</td>
<td>7313</td>
<td>723</td>
<td>20288</td>
<td>652</td>
</tr>
<tr>
<td>Neg</td>
<td>946</td>
<td>2547</td>
<td>41</td>
<td>2287</td>
<td>13</td>
</tr>
<tr>
<td>0 Bits</td>
<td>1911</td>
<td>1246</td>
<td>82</td>
<td>2,111</td>
<td>91</td>
</tr>
<tr>
<td>1 Bits</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2 Bits</td>
<td>7</td>
<td>1098</td>
<td>9</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>3 Bits</td>
<td>13</td>
<td>61</td>
<td>17</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>4 Bits</td>
<td>2435</td>
<td>407</td>
<td>127</td>
<td>2,776</td>
<td>113</td>
</tr>
<tr>
<td>5 Bits</td>
<td>4216</td>
<td>701</td>
<td>212</td>
<td>4,607</td>
<td>179</td>
</tr>
<tr>
<td>6 Bits</td>
<td>3646</td>
<td>443</td>
<td>102</td>
<td>3,223</td>
<td>93</td>
</tr>
<tr>
<td>7 Bits</td>
<td>2727</td>
<td>376</td>
<td>66</td>
<td>2,278</td>
<td>74</td>
</tr>
<tr>
<td>8 Bits</td>
<td>580</td>
<td>211</td>
<td>19</td>
<td>404</td>
<td>19</td>
</tr>
<tr>
<td>9 Bits</td>
<td>1031</td>
<td>92</td>
<td>4</td>
<td>823</td>
<td>4</td>
</tr>
<tr>
<td>10 Bits</td>
<td>1313</td>
<td>79</td>
<td>1</td>
<td>1,248</td>
<td>1</td>
</tr>
<tr>
<td>11 Bits</td>
<td>336</td>
<td>45</td>
<td>39</td>
<td>508</td>
<td>35</td>
</tr>
<tr>
<td>Percent</td>
<td>41.08%</td>
<td>20.20%</td>
<td>47.44%</td>
<td>38.98%</td>
<td>42.81%</td>
</tr>
</tbody>
</table>

**Table 5.** Most significant bit
Bibliography


[2] Lattner, C.
URL https://llvm.org/

URL https://github.com/riscv/riscv-isa-sim


URL https://github.com/riscv/riscv-gnu-toolchain

URL https://github.com/riscv/riscv-pk


