Code Generation Techniques

For Emerging Capability Architectures

by

Jacob Abraham

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved October 2022 by the
Graduate Supervisory Committee:

Michel Kinsy, Chair
Kevin Rudd
Andy Glew

ARIZONA STATE UNIVERSITY

December 2022

ABSTRACT

Memory safety and security issues continue to plague modern systems and are rapidly becoming a top priority. Capability architectures are a proposed solution that solve the problem at a fundamental hardware level, with several commercially viable options under active development. These new and evolving designs place higher demand upon the software tools needed to develop software to ensure correct execution. Capabilities introduce ideas that challenge typical architecture assumptions about the representation of data and its location in memory. This calls for a new core system software ecosystem.

A fundamental component of any software ecosystem is a compiler. Without a compiler, large critical components of the ecosystem must be written in assembly language; a tedious and possibly error-prone task. A compiler for a capability architecture that emphasizes memory security must above all else ensure functional and correct code generation, raw performance and power efficiency are no longer the chief concerns. Compilers for these architectures have been developed, but as capability architectures mature in complexity new compilation support is required. A set of techniques that help solve the compilation challenges for a capability architecture are presented in this work. These capability-aware compiler ideas are presented in their generalized forms to enable their adoption in other architectures and future extensions.

Some of the ideas presented come out of work on a compiler for a new capability architecture, Zeno. The Zeno compiler utilizes the extensible RISC-V instruction set and adds a set of global memory extensions, xBGAS (Extended Base Global Address Space), which is used to provide memory security. The Zeno compiler is described in detail as an implementation of the generalized capability-aware compiler. Static

analysis is used to evaluate the generated assembly code produced by the compiler. The generated code is sufficient to enable further testing of the Zeno architecture and drive its development.

ACKNOWLEDGMENTS

I would first like to acknowledge my thesis advisor, Dr. Michel Kinsy, for all his support through my degree. He helped make this thesis possible. Thank you to both Dr. Kevin Rudd and Mr. Andy Glew for their support. They both serve as members of my committee and have helped me develop professionally. Thank you to Alan Ehret and Mihailo Isakov who have worked along side me for many hours as all three of us worked on our respective theses. Their guidance has been invaluable to this work.

A quick thanks to Dr. Seth Abraham, for helping me with editing and letting me bounce ideas of of him. A final thank you to my friends and family, who saw very little of me as I completed this work. Thank you for putting up with my long winded rants about compiler design and computer architecture. You all kept me sane.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

LIST OF CODE SNIPPETS

Code Snippet Page

Chapter 1

INTRODUCTION

1.1 Problem Statement

Buffer overflows are a common memory vulnerability that have been a well-documented issue for over 50 years [27, 3]. Yet Out-of-bounds Write [10] and Out-of-bounds Read [8] are still number 1 and 3 respectively on the top 25 CWE software vulnerabilities [1]. Other top tier vulnerabilities include Use After Free (number 7) [9] and Improper Restriction of Operations within the Bounds of a Memory Buffer (number 17) [7]. Despite decades of research, the frequency of memory vulnerability CVEs demonstrates that a practical and effective solution has not reached wide adoption.

Another example is the failure of software to check the necessary permissions to access a piece of data. A ubiquitous example is the difference between kernel and user memory of an OS; a user program may not arbitrary access kernel memory, it must use the given interface (system calls and drivers). This type of memory protection is done at a large granularity, such as with memory pages. There is low interest in the software community to actually implement some of these mitigations. The C standard attempted and failed to implement Annex K [25], a proposal to add bounds checking to the C language. The performance trade-offs of a software based approach were just not acceptable.

The computer architecture community has developed a number of hardware-based solutions in an attempt to address continuing memory safety challenges [37, 40, 13,

41, 31, 28, 14]. A hardware-based solution supports both strong security and low overhead. With hardware enforced security, all software executing on the system can be protected. Hardware support lowers overhead by performing memory checks faster than is possible in software by using dedicated circuitry off of the critical path. In some cases, platform support for memory safety means programmers can catch memory bugs earlier, as illegal memory operations may not compile [13]. Programmers must still be aware of memory vulnerabilities, as the system may still crash if invalid conditions occur at runtime. A hardware solution guarantees no data will be leaked or lost, regardless of the programmer.

Capability pointers solve many of the described problems by associating memory addresses with metadata that describe the access permissions of the memory reference. Capability Hardware Enhanced RISC Instructions (CHERI) is a well described architecture that implements and uses capabilities. CHERI provides a set of architecture primitives (ISA) for several architectures (currently MIPS, RISC-V, and ARM), along with the associated software stack (OS, standard libraries, compiler). CHERI capabilities were originally 256 bits and have since been compressed to fit in a 128 bit register. The 128-bit capability is an atomic unit, it cannot be split apart. The CHERI compiler and some language semantics require large rewrites to be able to handle this [6, 36, 12].

CHERI is not suitable for some use cases. For example, in the HPC space it is common to have distributed memory nodes. A compute node will have access to local memory and access to global shared memory connected across a network. CHERI capabilities utilize 64 bit address bits [37, 40]. A 5-level paging scheme utilizes 57 of those address bits to address 128 PB [17]. This is not sufficient for exascale computing. Another limit concerns the act of allocation and revocation. Processes can share

memory with other processes by allocating a capability to the data. Processes can then unshare that memory by revoking the capability. In the CHERI model capabilities store all information needed about their memory in an atomic unit. Revoking a CHERI capability requires sweeping the entire memory space for any references to the capability [41]. This is impractical at scale.

Another hardware solution is Zeno, a security-focused global shared memory architecture [15]. Zeno implements capability pointers as *Namespaces*. Namespaces enable secure memory accesses which support sharing, access control, and memory isolation. The Zeno architecture implements an extended addressing model to enable shared global memory [32]. Zeno extends this addressing scheme to use and manipulate Namespaces.

Zeno [15, 2] can be described more generally as a capability architecture with a layer of indirection for the metadata. The capability is separable, one part of the capability is the base memory address and the other part is an identifier for metadata stored in a separate structure. Revocation of a capability avoids sweeping the memory by invalidating only the local record of the metadata. Indirection for the metadata also allows a system designer to practically increase the amount of metadata stored without increasing the size of the capability.

Separable capabilities consisting of mutable memory addresses and immutable metadata references are not operated on together. The mutable memory address can be computed as a normal memory address. The immutable metadata reference is passed through the processor unchanged. These separable pieces remain semantically tied together to allow for valid memory accesses. This introduces source code compilation challenges. Baseline code generation challenges include instruction selection and register allocation; an optimizing compiler faces even more such challenges.

Capability architectures share a common theme, they are a challenge to compile code for [6, 36, 12]. It is a balancing act between generating correct, optimized code while supporting a large ecosystem of legacy software. This work describes a set of compiler techniques to compile code for a capability architecture similar to Zeno, using separable capabilities with indirect metadata. We describe these techniques generically, this allows them to be applied to other systems with similar characteristics.

Separable capabilities provide unique challenges above the normal capability compiler issues. As there is no hardware primitives or datapath for the full width of the capability, operations must be lowered to the correct datapath. The spit capability components are not independent, memory accesses require that they be *semantically paired*. Semantically paired operations have no explicit binary encoding or hardware enforcement, but failure to follow these rules results in memory access failures. Therefore, operations need to be represented as the full width operations for as long as possible before being lowered to the correct code.

1.2 Aims and Objectives

The primary goal of this work is to support software on architectures with separable capabilities. The unique nature of these architectures requires new ways of compiling code. A compiler for a split capability architecture should handle the complexities of the architecture transparently to the programmer. Such a compiler is described and implemented in this work. Given a programmer who has written a piece of software with no bad assumptions about the architecture(s) this code may run on, the compiler should produce correct code in all cases. In the event of poorly written code that if

compiled for a split capability architecture will potentially behave unexpectedly, the compiler should warn the programmer this is unsafe.

In this work we do not cover how capabilities are created, or at what scope they are used. The assumption is that some other software/hardware entity manages the creation and destruction of capabilities. The prime objective of the compiler is to *maintain* capabilities. The compiler passes along all the information needed to access memory and treats the mechanisms of memory allocation and access as a black box.

1.3 Summary of Work

This work will provide a more in depth summary of several capability architectures. First described is a generic capability architecture, abstracting away architectural and micro-architectural concerns. A survey of capability architectures follows, which characterizes them in terms of the described generic architecture. This is used to describe the generic compiler work.

Next described is a series of compiler techniques and transformations that compile code for a generic split capability architecture. We describe a hypothetical generic instruction set and its representation in the compiler framework. We enumerate some of the complications that arise with some of the higher level software semantics and source level primitives.

Lastly, an implementation of a subset of the compiler techniques is shown for a specific capability architecture, Zeno. We describe the set of compiler changes required to implement the Zeno compiler in LLVM [19], an extensible compiler framework. Provided is an evaluation of the code produced by the Zeno compiler.

Chapter 2

CAPABILITY ARCHITECTURES

A capability architecture is a computer architecture design that implements `capabilities`. On a non-capability architecture, memory is accessed and referred to purely by the memory address. A capability combines a memory address with metadata that describes the memory access. A capability architecture replaces bare memory addresses in the system with capabilities.

2.1 Capabilities

Figure 1 demonstrates the utility of a capability. In a classic stack smash attack [27, 22], an attacker will attempt to write past the end of a buffer to overwrite values on the stack. Variables are set above the return address of a function in a function frame. Attackers can rewrite the return address to an arbitrary value by writing past the end of a buffer. This allows the attacker to jump to any piece of executable code, gaining full control of a system or reading/writing forbidden data. Consider a capability to the same buffer defining a set of constraints for the buffer, such as the bounds. An attacker attempting to write past the end of the buffer using the capability will generate a hardware fault, protecting the system. This relies upon the hardware restricting or even removing non-capability memory operations. A hardware capability defense is independent of what the programmer writes and is guaranteed by the physical architecture, making a more secure system.

Consider another similar attack. This time, instead of just using the return address

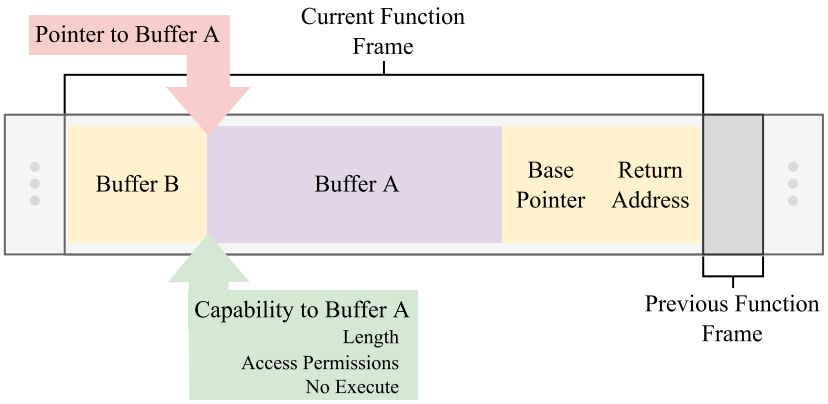


Figure 1. High Level View of Capabilities on the Stack

to jump to any piece of code, the attacker writes their executable code into the buffer using the capability. The attacker then finds an alternative way to overwrite the return address, the exact details of which is outside the scope of this example. An attack is still possible; to prevent this the capability is given access permissions. The access permissions state that the memory is not executable. The hardware enforces the access permissions. This allows for flexible permissions. One part of the stack can be marked non-executable to prevent attacks. Other parts can be marked as executable to allow for JIT code or self-modifying code, which some legacy systems may require.

This type of specific, fine grained permissions is not possible with current paging systems. Virtual memory, implemented with pages, has a number of permissions that can be set to protect the memory page [17]. However this can only be done at a page granularity, a capability has an arbitrary range to which it applies. This provides flexibility to system designers and engineers while still providing security.

The exact format of capabilities is implementation dependant, with various design trade-offs. Capabilities can be implemented on top of existing architectures, or as entirely new architecture descriptions. We discuss a generalization of two

different approaches to capabilities, followed by a discussion of multiple concrete implementations.

2.2 A Generic Capability Architecture

Consider a generic computer architecture. The architecture defines a set of operations that work on values stored in registers and another set of operations that provide memory access, a RISC-like load/store architecture. The architecture defines a single general purpose base register file. The exact implementation details of the architecture are unimportant. This description only focuses on the ISA level description, this is the typical view from compilers and other software point.

We will define capabilities on the generic architecture. An important component of the capability is a tag bit [40]. The tag bit is set by the hardware and enforces the immutability of the capability. This bit is not accessible by software. Tag bits prevent users from arbitrarily creating and/or modifying capabilities.

We describe two types of capabilities this system could be augmented with. We differentiate between the two as being `atomic` and `non atomic`. Atomic in this context refers the unified nature of a capabilities. Atomic capabilities are a single unit, they cannot be separated. Non atomic capabilities can be split and exist separately in the system. A valid memory access requires them to be brought together, but they are considered separate pieces.

2.2.1 Atomic Capability

On atomic capability architectures, capabilities are a single unit. They must be moved throughout the system as a single unit. Capabilities cannot be changed by normal instructions, only special capability instructions can operate on them.

Single unit capabilities cannot be split apart; they must be stored in memory and registers together. This can be implemented without modifying an existing architecture, as many virtual addressing schemes do not utilize all of address bits. The unused bits can be co-opted for use as a limited form of capabilities. An architecture designed for single unit capabilities may define a separate capability register file, with some advantages. Separate capability registers can be properly sized for capabilities independently from integer registers. This also separates integer data and memory addresses, which may have further security advantages.

Separate capability registers requires duplicating instructions. For example, an `add` instruction. The architecture must define two versions, one for bare integers and one for capabilities. This is required for pointer arithmetic operations.

2.2.2 Non Atomic Capability

On non atomic architectures, the base general purpose registers and their associated instructions can remain unchanged. A separate set of register is implemented that will store the metadata. The exact method of implementing this is entirely implementation dependant. The full capability is effectively a super register consisting of a base general purpose register and a metadata register. This is similar to x86 general purpose registers that are extended for wider bit widths [17]. This allows the instruction

set to encode registers as a single value. Another approach is have entirely separate metadata registers. These registers are encoded in instructions explicitly. This is similar to x86 segment registers [17]. Either way, memory addresses and metadata are stored separately. Bringing the memory address and the metadata data together creates a valid capability.

No new operations need to be defined for the metadata registers. Immutable metadata is either created or passed between registers. Instructions that modify metadata are undesirable and break security guarantees. Existing instructions can operate on the base memory addresses normally. The resulting memory address may no longer create a valid capability. Attempting to access memory with it is handled by the hardware in an implementation dependant way.

This provides an opportunity for a number of optimizations. Consider an array of memory addresses that all point to the same region of memory, such as array of function pointers. A single piece of metadata can be stored for the entire array, allowing an application developer to compress the storage of capabilities in memory.

There are two subsets of the non atomic capability considered. Metadata stored directly in the metadata register is the obvious approach. A more careful design may use the metadata register as a reference to metadata stored elsewhere in the system. Adding this layer of indirection allows system designers to store more information in the metadata. Indirection also allows metadata to be reused. Capabilities now consist of two references, one to memory and one to metadata. Metadata indirection allows the metadata to be arbitrarily sized, subject to other constraints outside the scope of this paper, without having to modify the full size of the capability.

This opens up possibilities to scale capabilities across systems. In the HPC space it is common to have distributed memory nodes A compute node will have local

memory it can access, as well as global shared memory connected across a network. Capabilities pointers being applied to this system need the following attributes in addition to the typical pieces of metadata

- Notion of ownership, which node owns the capability
- Notion of sharing, which nodes is this memory shared with
- Provisioning, how do we tell the capability and other nodes about this capability.

Consider revocation, where the owner of a capability has previously shared it with other nodes in the system. After allowing those nodes access, the owner wishes to revoke the capability and disallow access. Revoking the capability requires only invalidating the local copies of the metadata. Any node using the existing capability may still have a local copy, but any memory access made will be rejected due to the invalidated metadata.

2.3 Fat Pointers

Memory tagging [30] is the procedure of using the upper bits of an address to store tags about the data. On x86-64 systems using 4 level paging, virtual addresses only use 48 bits of the full 64 bits possible. With 5 level paging, this is increased to 57 bits [17]. Both systems have a handful of bits that are defined to be the same as the most significant bit of the address, this is a requirement set by the processor. An OS can set these upper bits to a tagged metadata value, and then before doing a page table walk reset them to their proper value [30]. The tagged metadata can be used for multiple purposes, but in the context of a memory-safe architecture the primary use is to make security guarantees. For example, the tag could contain a reference

count that would prevent a use after free vulnerability. Tagged pointers allow an OS to support a limited form of capabilities inside of existing registers.

Tagged pointers are a technique that is entirely software independent and works with existing hardware. Existing software can utilize tagged pointers without changes or recompilation, as the capability pointer fits within a normal 64-bit pointer. New hardware does not need to be developed, as the capability pointers fit within the existing registers. All the heavy lifting is done by the custom OS which supports memory tagging. The OS supplies memory allocation functions which set the memory tag bits and translates the memory address to the proper virtual address before passing the address to the MMU. The disadvantages of this technique are the limited set of addresses that are possible and that the tag bits are unprotected. 5 level paging can address 128 PB of memory [17], but this cannot meet the modern exascale requirements of HPC systems. Furthermore, 5 level paging only leaves 7 bits to tag memory with. It is possible to leverage some of the low order bits. An architecture may require memory to be aligned to a certain width. For example a 64 bit integer that is required to be 64 bit aligned has an memory address with 2 free low order bits. Tags can be formed from both the extra bits left over from virtual addressing and aligned memory. The small handful of bits available severally limits the amount information that a tag can hold; and without hardware enforcement tags are unprotected. Tagged pointers have previously been implemented on 64 bit pointers in existing registers [5].

Memory tagging can be implemented transparently and with no changes to the hardware, requiring only a custom OS. This is advantageous as the cost of software is significantly lower than for hardware.

Fat pointers are a logical extension of memory tagging. The handful of bits left over from virtual addressing is not enough to encode more complicated metadata. Fat

pointers make addresses larger to accommodate more metadata. We can consider memory tagging to be a limited subset of fat pointers. Because a fat pointer can be defined architecturally to be much bigger than a normal memory address, more access control and permissions are possible.

Memory tagging and fat pointers can be classified as an atomic capability, they are not separable. The entire capability is stored in a single register and used as a single unit throughout the system. This type of capability has been implemented in software on top of existing architecture. Architectures have also been custom designed for this.

2.4 CHERI

CHERI [37, 39, 35, 41] is a prolific implementation of a capability architecture using fat pointers. CHERI introduces a capability co-processor on top of a base ISA. The co-processor has capability registers, which store a 128-bit compressed capability. These capabilities contain both the address and the metadata in a single atomic unit, with special capability instructions to utilize them. This replaces typical memory access instructions in the base ISA. CHERI treats its capabilities as atomic units, changing the size of a pointer. This requires the mentioned hardware changes, as well as a new OS and compiler. With these CHERI specific tools, a generic piece of software can be run with capabilities.

In recent years, CHERI has emerged as a strong candidate for a realistic capability architecture. CHERI has a stable architecture on top of RISC-V, previously MIPS, with a relatively mature software stack to accompany it. Currently in prototype production for evaluation purposes is Morello, an ARM based processor enabled with

CHERI extensions [34, 16]. This is a commercially viable capability architecture product. Morello is proving to be both a secure and stable system that is drawing attention across the industry.

As mentioned, CHERI falls into the atomic capability architecture category. The CHERI co-processor defines a full set of capability operations on a completely separate set of capability registers. This provides certain advantages, as arbitrary software may run on a CHERI system without capabilities, and utilize capabilities for certain parts. This can significantly lower the barrier to entry for systems wishing to introduce hardware based security.

2.5 MPX

Memory Protection Extensions (MPX) was a set of architecture extensions to the x86 architecture to provide capabilities [26]. MPX required substantial changes across the software stack. At its core, MPX introduced instructions and registers to perform bounds checking on a pointer. These bounds were stored in a bounds table, which had to be managed by the operating system. The actual bounds checking instructions have to be added by a MPX aware compiler, which also links in a MPX runtime library.

MPX is an example of a capability architecture gone wrong. It focused almost solely on bounds checking and the actual implementation left much to be desired. The changes required for an OS to be able to support MPX and the large impact on compiler optimization opportunities was a deal breaker. MPX code generated by the compiler frequently needed many optimizations disabled to be stable. This by itself was not a feature of MPX, but rather a factor of short development. MPX

could also not support certain C idioms and introduced potential race conditions in multi-threaded code. The biggest issue ultimately came down to performance, MPX destroys most performance characteristics. There are many more instructions to execute, introducing huge bottlenecks into the out of order pipeline. MPX also trashes the cache, lowering memory performance. Besides the performance issues, MPX in many cases did not even succeed in doing proper bounds checking [4, 26].

All of these reasons led Intel to remove support for MPX in later processors. The most important lesson from MPX is that a capability architecture needs two things. For one, it must be performant. The computer architecture and software development communities are not tolerant of security at the expense of performance. Secondly, the large number of software changes required severely damaged MPX.

2.6 Zeno

Zeno [15] is a capability architecture that supports an extended addressing model to create data-center scale global shared memory. Memory beyond the local base 64-bit address space is accessed with extended 128-bit addresses. The xBGAS RISC-V ISA extension defines the encoding of load and store instructions to the extended address space [32]. Zeno utilizes these additional ISA extensions to provide a memory abstraction known as a Namespace.

Namespaces can be attributed to the non atomic capability model. The base memory address is stored in a general purpose register, while a Namespace Identifier (NSID) is stored in an extended register. These two together make up a memory reference into a Namespace. The Namespace consists of metadata stored in memory

that describes the memory region. This provides a single point of storage for a region of memory, with many NSIDs then pointing to the Namespace.

The Namespace is treated as immutable, once it is created it is not changed. A reference to it can be revoked to provide scalable memory security. This immutability means software cannot modify the NSID. The NSID must maintain its association with the memory address to ensure correctness. The immutability is enforced by the hardware. The association is enforced by software. Software has a great deal of flexibility on how to handle Namespace storage and management, while still enforcing security.

We classify Zeno as a non-atomic capability architecture. Capabilities can be split apart and remain valid.

Chapter 3

COMPILER DESIGN

We describe common features in modern compilers. We use LLVM [19] as a concrete example to simplify later discussions of compiler techniques.

3.1 Compiler Pipeline

At a high level, a compiler transforms human readable source code into machine readable code through one or more transformations. These transformations are called passes, they apply some action to an input and emit something as an output. An overview of the compiler pipeline is in Figure 2. The compiler can be logically broken down into three sections, with some exceptions. The most important concept is the Intermediate Representation (IR).

The IR is the common language all components of the compiler speak. Between the front end and the middle end is a conversion to the IR; between the middle end and the back end is a conversion from the IR. A common IR allows the compiler to be

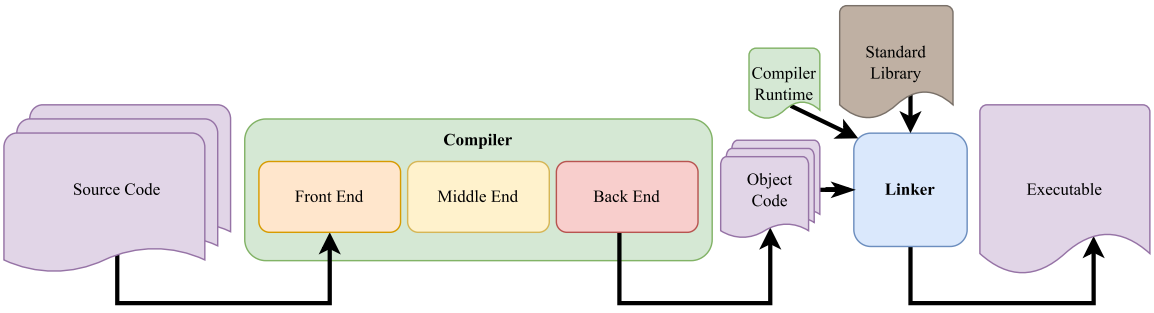


Figure 2. Compiler Pipeline

easily retargetable. A compiler is able to reuse much of the middle end for multiple source languages and target architectures by doing most of the optimization work in the IR.

The front end of the compiler is responsible for converting the source language to a machine readable format. This parsing results in an abstract syntax tree (AST), a formal representation of the source. The AST contains all the information to do front end checks, such as semantic analysis and type analysis. This information is used to inform correct IR generation.

The IR generated by the front end is fed to the middle end. The bulk of the work done by compiler occurs in the middle end, with heavy analysis and optimization passes. Common optimizations can be done independently of the target architecture and improved with architecture dependant information. Many optimization passes operate on the IR in SSA form [29, 11]. An important component of this is the usage of virtual registers. Virtual registers are analogous to a variable in a high level program, except they are representative of a physical register on an architecture.

The role of the backend is to emit the IR as machine code. This involves register allocation, the act of changing virtual registers into physical registers. This also includes target architecture specific optimizations that either only apply to machine code, or can only be done at this stage. The act of converting IR to machine code is instruction selection and the order of these instructions is determined through instruction scheduling.

Another component of note is the linker and associated runtime libraries. The linker takes many pieces of compiled code and combines them into a larger archive of code, or into a format capable of being executed on a given OS. This involves a number of relocations. When the compiler is compiling code, it often does not

know the address of other components in this system, such as function calls or global variables. These are encoded in the machine code object format as relocations, which are just a way of telling the linker that it needs to fill on the correct address here before the code is executable. Relocations can also be references to library code supplied by the compiler (runtime library), included as apart of the language standard (standard library), or included by the user.

All of these pieces can be found in a compiler for any system, in one form or another.

3.2 LLVM

LLVM [19] is a compiler framework built for extensibility. Originally built as a research project, it has become an industry standard for building compilers. LLVM IR is a core component of the framework, it is a retargetable intermediate representation for LLVM. This allows any language that has a front end that emits LLVM IR to be run on the target. A developer supporting a new architecture defines a set of instructions, a set of rules on how to express the IR using these instructions, and a set of rules for how to encode these instructions. This constitutes a complete compiler for the new architecture. Passes can be added at all levels of the compiler to provide new optimizations or enable new code generation. Multi-Level Intermediate Representation (MLIR) [20], is a new component of LLVM that improves the extensibility of LLVM. We do not focus on it in this work.

LLVM is primarily written in C++, but many of the key components of the backend are written in *TableGen*. TableGen is a domain specific language (DSL) built specifically for LLVM to describe back ends. Developers write an architecture

definition declaratively and TableGen generates the boilerplate C++. One of the most significant generated components is the instruction selector. TableGen generates a table that contains a simple set of operations. The instruction selector can then use the operations encoded in the table to perform complex instruction selection. In the event that TableGen cannot express the instruction pattern, a fallback path allows developers to hand code the C++ required to select the instruction. Another advantage of the TableGen definitions is that they allow the encoding of instructions to be almost entirely defined from just the TableGen. Therefore an assembler, disassembler, and associated tools can be automatically generated for a target.

Below is a brief discussion of the front and middle end; followed by a more in-depth discussion of the backend. This is by no means a complete description of the full LLVM framework, but it provides a frame of reference for the rest of the compiler discussion in this work.

The front end consists of a parser and a series of passes. The parser converts the source language to an AST containing all the semantic information. A series of *Sema* passes perform semantic analysis on the AST. LLVM defines the typing system of a language as a part of Sema. Finally, the AST is converted to LLVM IR in a process known as *CodeGen*.

The middle end contains many optimization passes that operate on the LLVM IR. These passes are all IR to IR passes, and are generally enabled/disabled based on command line flags. For x86 with the optimization flag $-O3$ passes, over a hundred optimization passes are run in LLVM 13.0.0.

We will explain components of the back end in greater depth, as this is highly relevant to further capability compiler discussions. In the back end, the LLVM IR for a function is converted to multiple SelectionDAGs. Each SelectionDAG is a DAG rep-

resentation of basic block, a sequence of instructions with a single entry and exit point. Instructions are represented as Independent SelectionDAG (ISD) Nodes, which can be generic target agnostic nodes or nodes for specific architectures. The construction process and optimization of the SelectionDAG is largely architecture independent. The conversion processes from IR to SelectionDAG can generate operations not defined for the target architecture. Operations supported on the target architecture are *legal*, operations not supported at *illegal*. The SelectionDAG legalizer converts illegal nodes to legal ones. The DAG is iteratively combined, optimized, and legalized. This produces a set of optimized nodes valid for the target architecture. The legalized DAG is then *selected* to a DAG of Machine Instructions (MachineInstr). Machine Instructions are an internal LLVM representation of the final instructions to be emitted. The DAG of Machine Instructions is *scheduled* to provide a linear ordering of the code. A series of optimization passes, both target independent and dependant, transforms the Machine Instructions. The register allocator runs during this process, converting the virtual registers to physical ones. Finally, the optimized Machine Instructions are emitted as Machine Code Instructions (MCInst), a representation closer to the final binary encoding. LLVM performs a series of passes on these instructions that result in the final assembly being emitted from the compiler.

Chapter 4

COMPILATION TECHNIQUES

We discuss generic capability architecture compilation challenges in LLVM [19] and how they can be solved in LLVM. These ideas are not specific to any one compiler. LLVM is an industry strength compile framework commonly used in both industry and academia due to its high degree of extensibility and sharing common optimizations between architectures and languages. For this reason, much of the work will be explained in the context of LLVM's constructs and framework.

The proposed capability architecture with separable memory addresses and metadata introduces significant compiler challenges. Compilers commonly make two assumptions about pointers, which do not hold for this kind of architecture. One assumption is that a pointer is the same size as the largest native integer type on the platform. Consider x86-64, with 64 bit registers. The largest possible integer stored in these registers is 64 bits and the largest possible memory address is 64 bits. A capability architecture with separate integer register and capability registers has different size constraints. A memory address is the width of the capability register, which may be larger than the base integer register. This assumption caused the CHERI compiler to rewrite and fix large parts of the core LLVM framework [12]. The other assumption is that pointers can be directly represented on the target architecture, which is to say that a pointer is stored in a single register. Architectures like CHERI still have capabilities stored in a single register. In our proposed separable model, capabilities are split between multiple registers with asymmetric operations on the

registers. With both of these assumptions broken, existing compiler techniques relying on them will produce incorrect code for this architecture.

We will demonstrate a few key examples from the source of LLVM, which showcase these assumptions. One of these such places is for the generation of target specific info in the code generation portion of clang, LLVM's front end for the C family of languages. In the target specific section for RISC-V, setting the bit width of registers and other architectural features is done with the call to `getPointerWidth`. This assumes that a pointer is the largest integer. This code is shown in context in Code Snippet 1.

Another poor assumption is made in the code generation portion of LLVM itself. This time involving an optimization for switch statements. LLVM uses the pointer type to determine if an operation is legal (Code Snippet 2) and also to compute a potential bit width for a bit mask (Code Snippet 3). Both of these assumptions break when capability pointers are not the size of a integer.

Patching these poor assumptions is a critical part of developing a compiler for a capability architecture. The rest of this chapter will describe a generic instruction set for separable capability architecture, followed by a description of the techniques used to generate code.

4.1 Instruction Set

We describe a generic, RISC-like architecture with the symbolic name Alpha. We assume that this is a mature architecture with a fully functioning compiler, implemented in LLVM. We describe a generic capability architecture with the symbolic name Beta. Beta is an extension to Alpha. We keep Alpha and Beta generalized; no

```
case llvm::Triple::riscv32:
case llvm::Triple::riscv64: {
  StringRef ABIStr = getTarget().getABI();
  unsigned XLen = getTarget().getPointerWidth(0);
  unsigned ABIFLen = 0;
  if (ABIStr.endswith("f"))
    ABIFLen = 32;
  else if (ABIStr.endswith("d"))
    ABIFLen = 64;
  return SetCGInfo(new RISCVTargetCodeGenInfo(Types, XLen, ABIFLen));
}
```

Code Snippet 1. clang/lib/CodeGen/TargetInfo.cpp:11165 LLVM 13.0.0

```
EVT PTy = TLI->getPointerTy(*DL);
if (!TLI->isOperationLegal(ISD::SHL, PTy))
  return;
```

Code Snippet 2. llvm/lib/CodeGen/SwitchLoweringUtils.cpp:285 LLVM 13.0.0

```
const int BitWidth = TLI->getPointerTy(*DL).getSizeInBits();
assert(TLI->rangeFitsInWord(Low, High, *DL) && "Case range must fit in
    bit mask!");
```

Code Snippet 3. llvm/lib/CodeGen/SwitchLoweringUtils.cpp:391 LLVM 13.0.0

specific bit widths are described and the actual microarchitectural mechanisms are out of scope for this work. We provide an ISA-level description of both Alpha and Beta.

Alpha defines a set of common arithmetic operations, a set of common bitwise operations, a set of common control flow instructions, and a set of common memory operations. These operations perform work on a set of registers of K-bits. For notation purposes, Alpha instructions will be written as `INST` and Alpha registers will be written as `A#`. The destination register is written first for most instructions. For example, multiplying register 2 and 3 into register 4 is written as `MUL A4, A2, A3`. A copy instruction from register 5 to register 7 is written as `COPY A7, A5`. A load instruction to register 1 with the memory address in register 10 is written as `LOAD`

`A1, A10`. A store instruction from register 1 with the memory address in register 10 is written as `STORE A1, A10`.

Beta is now implemented on top of Alpha. Beta redefines the registers to be of N bits, with 2 sub registers. The first sub-register corresponds to Alpha's K-bit register. The second sub-register can be a reference to a piece of metadata, with a width of N-K bits. For notation purposes, we write the full N-bit register as `B#` and refer to the sub-registers by zero-based indexing as `B#(#)`. For example, to refer to `A17` of Alpha we write `B17(0)` and to refer to the second sub register as `B17(1)`. It is equivalent to refer to either the `A` register or the zero index `B` register, they are aliases of each other. Alpha instructions do not take the full register as operands, the only valid operands are ones of the form `B#(0)`. There are three exceptions to this, for `COPY`, `LOAD`, and `STORE`. For `COPY`, any combination of register operands is valid. Copying a sub-register to a full register is similar to a vector insert and copying a full register to a sub-register is similar to a vector extract. For `LOAD` and `STORE`, the second operand *must* be the full `B` register. The source/destination register can be the full `B` register or one of the sub-registers. The registers are summarized in Figure 3.

The `A`/`B(0)` registers are used for normal computation. When forming a full capability, the `A`/`B(0)` registers are used for memory addresses and the `B(1)` registers are used for metadata references. The full `B` register containing the full capability is not accessible by software. Specifying it as an operand for a copy instruction is a shorthand for assembly programmers.

For both Alpha and Beta, we define a simple example calling convention.

- Register 0 is the return address
- Register 1 is the stack pointer

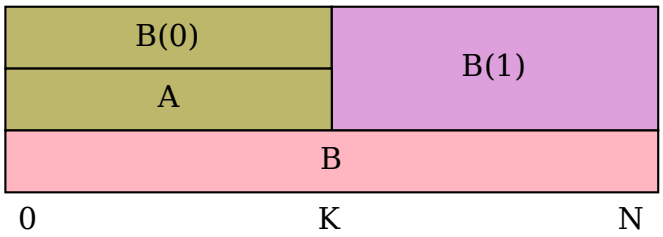


Figure 3. Registers for Alpha and Beta

- Register 3 through 9 are saved registers that any callee must save
- Registers 10-... are temporary registers are not guaranteed to be saved by the callee
- Arguments are passed in registers 10 through 20
- Register 10 is the return value

When a load or store occurs, the hardware will use the full `B` register as a capability. `B(0)` holds the memory address and `B(1)` holds the metadata reference. If the memory operation is invalid, the hardware will trap.

As an example, we hand write some C code in assembly code for both Alpha and Beta. Code Snippet 4 shows a single offset used to get a pointer from a double pointer. That new pointer is used to load more memory from the same offset, which is then added to the pointer to create a return value. The Alpha code in Code Snippet 5 is quite simple, we calculate an address and use it to load the new pointer. We now need to save this new pointer to later add it with an offset, but the pointer needs to also be used to calculate the next address. The Alpha code looks very similar to what a compiler for a RISC-like architecture such as MIPS or RISC-V might produce. Code Snippet 6 is a bit more complex. The same core operations are done, but because we need to return complete capabilities that are preserved after pointer arithmetic, there are two additional copy instructions. The copy instructions are explicitly required

```
int64_t* foo(int64_t** x, int64_t off) {
  return x[off] + x[off][off];
}
```

Code Snippet 4. Example C Code

```
foo:
  ;; calculate x[off]
  SHL A11, 3
  ADD A10, A10, A11
  LOAD A12, (A10)
  ;; calculate x[off][off]
  ADD A13, A12, A11

  LOAD A13, (A13)
  ;; compute sum
  ADD A10, A12, A13

  RETURN
```

Code Snippet 5. Example Alpha Code

```
foo:
  ;; calculate x[off]
  SHL B11(0), 3
  ADD B10(0), B10(0), B11(0)
  LOAD B12, (B10)
  ;; calculate x[off][off]
  ADD B13(0), B12(0), B11(0)
  COPY B13(1), B12(1)
  LOAD B13(0), (B13)
  ;; compute sum
  ADD B10(0), B12(0), B13(0)
  COPY B10(1), B12(1)
  RETURN
```

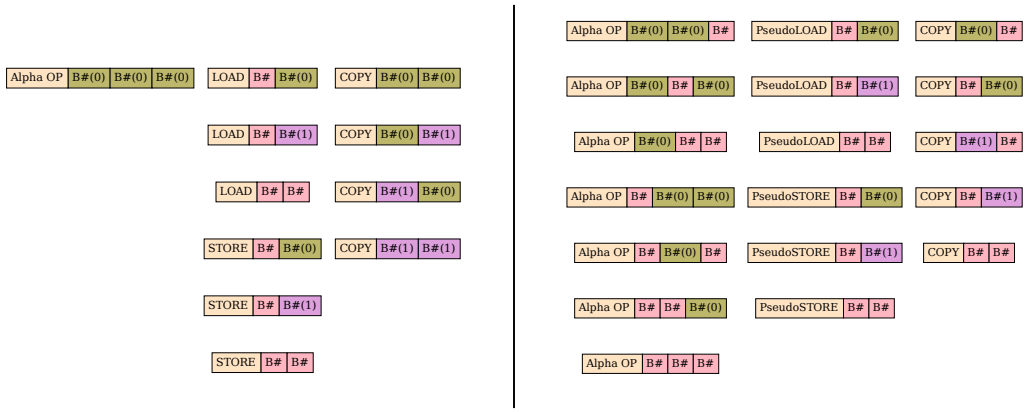
Code Snippet 6. Example Beta Code

because the computation on the memory address does not implicitly imply a metadata copy.

4.2 Pseudo Instruction Set

We have described the ISA of a generic capability architecture. The instructions can be directly encoded into the compiler. Passes can select the IR to the instructions. This is the normal path of compilation.

A simple technique to streamline compilation is defining pseudo-instructions. Pseudo instructions allow a compiler to select higher level instructions. These map to a set of real instructions for the hardware. This is particularly useful in a RISC-like architecture, where instructions do simple operations. For example, on a RISC-like architecture accessing a global variable in position independent code is a multi-step



(a) Real (b) Pseudo

Figure 4. Beta Instructions

operation. First the PC is loaded into a register and an offset is used to compute the address of the global variable. The a memory operation can be executed using the computed address. An instruction selection pass in the compiler will have a hard time correctly generating these instructions. Instead, the instruction selector can generate a single pseudo instruction. The pseudo instruction then passes through the optimization pipeline like normal before being expanded at the last possible second, right before the final assembly is emitted. This simplifies the instruction selector. It also simplifies many of the optimization passes, as they only need to maintain the integrity of one pseudo instruction instead of many real instructions.

The compiler for the Beta architecture implements a wide selection selection of pseudo instructions. Loads and stores on the Beta architecture use a N bit capability. Since there do not exist any real N bit registers or real instructions that can use N bit addresses, we introduce pseudo registers and pseudo instructions. The full registers in Beta are declared to be pseudo registers. Any instruction can take a pseudo register as an operand in place of a real physical register operand. This makes the instruction a pseudo instruction, which is expanded to a real instruction later. `B#(0)` and `B#(1)` are

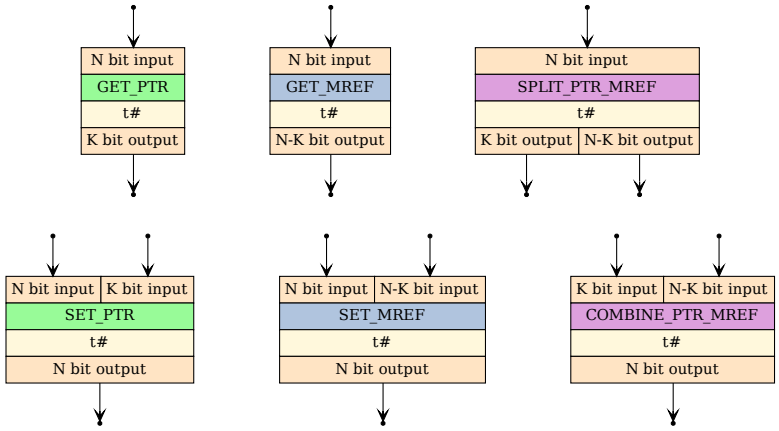


Figure 5. DAG Mapping Operations

real registers and `B#` is a pseudo register. The pseudo register `B#` can be the argument or result of any instruction. For `LOAD` and `STORE` we define explicit pseudo instructions, `PseudoLOAD` and `PseudoSTORE`. This is summarized in Figure 4a and Figure 4b, which enumerate all real and pseudo instructions, respectively. Since we are just talking about Beta instructions, we use the Beta register names. Alpha instructions can use Beta registers, as `A` and `B(0)` are aliases of each other.

Introducing pseudo instructions allows other parts of the compiler to be simpler. Instructions are not expressed with the full complexity of the ISA and are later converted to real instructions. To better express high level intent, we can include extra arguments as hints to `PseudoLOAD` and `PseudoSTORE`. This disables the register allocator for a particular instruction. The extra instructions to ensure correctness can be automatically included and the remaining registers can be assigned using the register allocator. This provides greater flexibility in selecting instructions.

4.3 Capability Mapping

To describe the mapping of pointers to a capability architecture, the following stage of the compiler is described. LLVM IR is transformed to target specific machine code in the following steps.

1. Build an initial DAG from IR
2. Optimize and legalize the DAG
3. Select machine instructions from the DAG
4. Schedule the machine instructions from the DAG
5. Optimize the machine instructions
6. Register Allocation
7. Optimize the machine instructions
8. Emit the final machine code

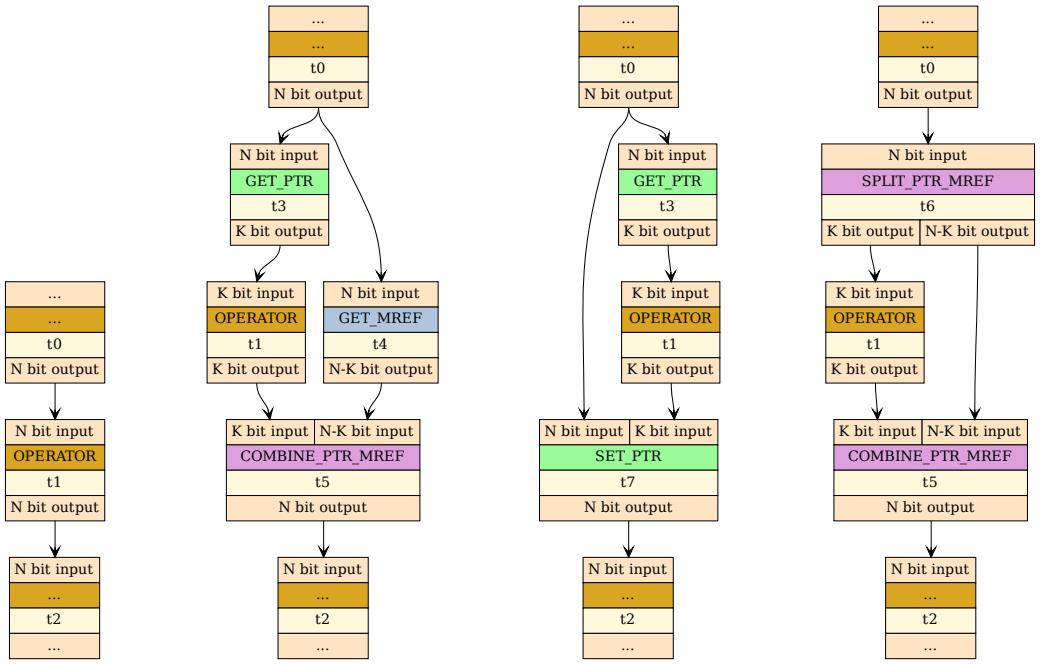
The mapping will focus specifically on the DAG. A DAG is a directed, acyclic graph and is used to represent a piece of control and data flow through a basic block in a compiler. A basic block is a piece of code with a single entry point and a single exit point. In LLVM, one very important property of the DAG is that it may contain both target independent nodes, target dependant nodes, and machine instruction nodes. Initially it contains mostly target independent nodes. Through legalization and optimization those target independent nodes may get mapped to target dependant nodes, or even machine instructions. Target dependent nodes can also be mapped to machine instructions. The selection phase (step 3) is where target independent nodes and target dependent nodes must be mapped to machine instructions, as past this stage all nodes must be machine instructions so they can be scheduled.

In a generic compiler framework such as LLVM, the DAG is initially created via a target independent pass. The DAG is constructed with many potentially *illegal* nodes. It is the role of the legalizer to define what operations are valid for the target architecture and what operations need to be converted into a series of one or more nodes that are supported. Since the legalizer for Beta is built on top of Alpha, we assume most of the work to be done for us. The Beta legalizer just needs to define what is different from Alpha. We define the size of an integer for Beta to be K bits and the size of a pointer to be N bits. For memory operations, the memory address operand MUST be a full N bits as this is the valid capability. The value operand, either the destination value or the source value, can be of either N bits or of K bits as both are valid instructions for Beta. Any remaining N bit wide instructions are NOT valid on Beta, as Beta defines no new instructions to operate on N width. Beta only uses the K bit operations from Alpha. To legalize instructions for Beta, we *map* these instructions to their K and N-K bit equivalents to create valid instructions. We call this *DAG mapping*.

To map and make valid instructions, we define a set of six operations to manipulate the DAG.

- *GET_PTR*
- *GET_MREF*
- *SET_PTR*
- *SET_MREF*
- *SPLIT_PTR_MREF*
- *COMBINE_PTR_MREF*

GET_PTR and *GET_MREF* take a N bit operand and extract out a K bit and N-K



(a) The original DAG

(b) Wrapped in full capability preservation

(c) Wrapped in minimal capability preservation

(d) Wrapped in alternative capability preservation

Figure 6. Possibilities for Capability Preservation

bit result, respectively. *SET_PTR* and *SET_MREF* take a N bit operand and a K bit and N-K bit result, respectively, and insert the second operand into the first. *SPLIT_PTR_MREF* takes a N bit operand and produces a K bit result and a N-K bit result. *COMBINE_PTR_MREF* takes a K bit operand and a N-K bit operand and combines them into a N bit result. This operations are summarized graphically in Figure 5.

We define a number of heuristics to apply the operators to legalize the DAG Many are simple substitutions, for example a truncation from N bits to K bits. This is not a valid sequence of instructions, as it is not valid to truncate a capability pointer. For the theoretical Beta architecture, it is valid to extract just the memory address and

operate on it. So a truncation can be modeled as a `GET_PTR`. As another example, an N bit constant is not valid to write in a source program. But because it is used as an operand to a N bit instruction, the initial DAG creation processes will generate N bit constants. These are trivial to handle as we can just rewrite them as either K bit or N-K bit constants, depending on the context.

We will now consider five cases where we apply our heuristics.

1. No operands and one N bit result value
2. One N bit operand and no result value
3. One N bit operand and one N bit result value
4. Two N bit operands and one N bit result value, and the operation is a 'decision'
5. All other cases

For these cases, we apply them to almost all instructions, except for a handful of special cases. These include instructions like loads and stores, constants (we handle them specially, see above), inline assembly (we do not want to change what the programmer has written), function calling code, and instructions that have already been selected. The first case and last case can both be handled trivially. To handle the first case it is correct to just rewrite it as a K bit or N-K bit operation. Since any of other transformations will take these as operands, it is safe to just rewrite the result type. The last case is also trivial to solve, we ignore it. When counting the number of operands, we do not include constants as an operand.

Solving the second and third cases is similar, as the second case is half of the solution to the third case. The solution to this is simple but perhaps foreign, it does however make intuitive sense in the pointer arithmetic case. Say we are running a loop through a simple array, and we have a capability pointer to the first element. We want to add a value to the memory address to go to the next pointer and just pass

along the metadata reference value. We can generalize this for all operations, and call these types of transformations 'capability preservation'. We show an example DAG of this in Figure 6a.

Many possible combinations of the DAG operators create valid capabilities. We present three ideas for wrapping operations. The general idea is the same, extract out the pointer, transform it in some way, then re-wrap it. The verbose full wrapping, shown in Figure 6b, extracts the pointer and metadata reference in separate steps, operates on the pointer, and combines them again into N bits. We also present a combined approach of this in Figure 6d, which combines the extract steps into one node. Finally, we present a minimal approach in Figure 6c that just extracts out then pointer and then reinserts it into the original N bits. Other combinations of these operations is possible; these are the most representative of all the available functionality.

We primarily implement Figure 6b as it provides the simplest DAG representation that is easiest to optimize. Figure 6d is similar, but it contains fewer total DAG nodes. Although fewer in number, the DAG operators producing multiple outputs require more complex implementation effort and maintenance. Figure 6c also has fewer nodes and initially results in less register to register copies. A competent optimizer will recognize these extra register copies and coalesce them, nullifying the perceived advantage. As the full wrapping is easier to implement, this pattern is less used. Other potentially more optimal patterns can be created using these operations. The described operators are designed to be as general and flexible as possible while still being intuitive to use.

Any of the presented approaches produces valid capability code and is able to

handle both cases; one operand/one result and one operand/no result. For the no result case, trimming away the combining capability part produces a correct result.

Handling the remaining case is slightly different. This case revolves around what is essentially an if statement, if the comparison of 2 capabilities is true the result is the first capability, if the comparison is false the result is the second capability. We can actually handle this with two streams of execution. We duplicate the instruction to be wrapped, one handles the K bit memory address and the other handles the N-K bit metadata reference. Then we combine the results of the two streams of execution as result of the now transformed node.

To optimize these operations, we can feed the code as-is into standard compilation optimizations. As the number of wrapped operations grows, many of these wrappings end up being done on the same data. Wrapping and unwrapping individual nodes produces a large number of extra copies. So additional passes take sequential wrappings and merge them together.

We will walk through storing a four byte integer to a pointer at a given offset, which is essentially an array. Figure 7a shows the DAG prior to mapping. It is important to note that almost all bit sizes are initially set to N, as it is not valid to mix bit sizes in a DAG in the LLVM framework. t1 and t5 are both 32 bit outputs, as they are known inputs. t5 is valid 32 bit input for t6, but t1 is not a valid input for t3. We fix this by rewriting the t3 to have a 32 bit input, giving Figure 7b. N bit constants like t2 can automatically be lowered and since it is going to t3, which we already know needs a 32 bit input. Lowering t2 to 32 bits produces Figure 7c, note that several other bit sizes change as a result in t3 and t4. This is the exemplar case for DAG mapping, so we apply the DAG operators and get Figure 7d. This leaves the DAG with one inconsistency, a 32 bit output from t3 going to t4. To make a valid

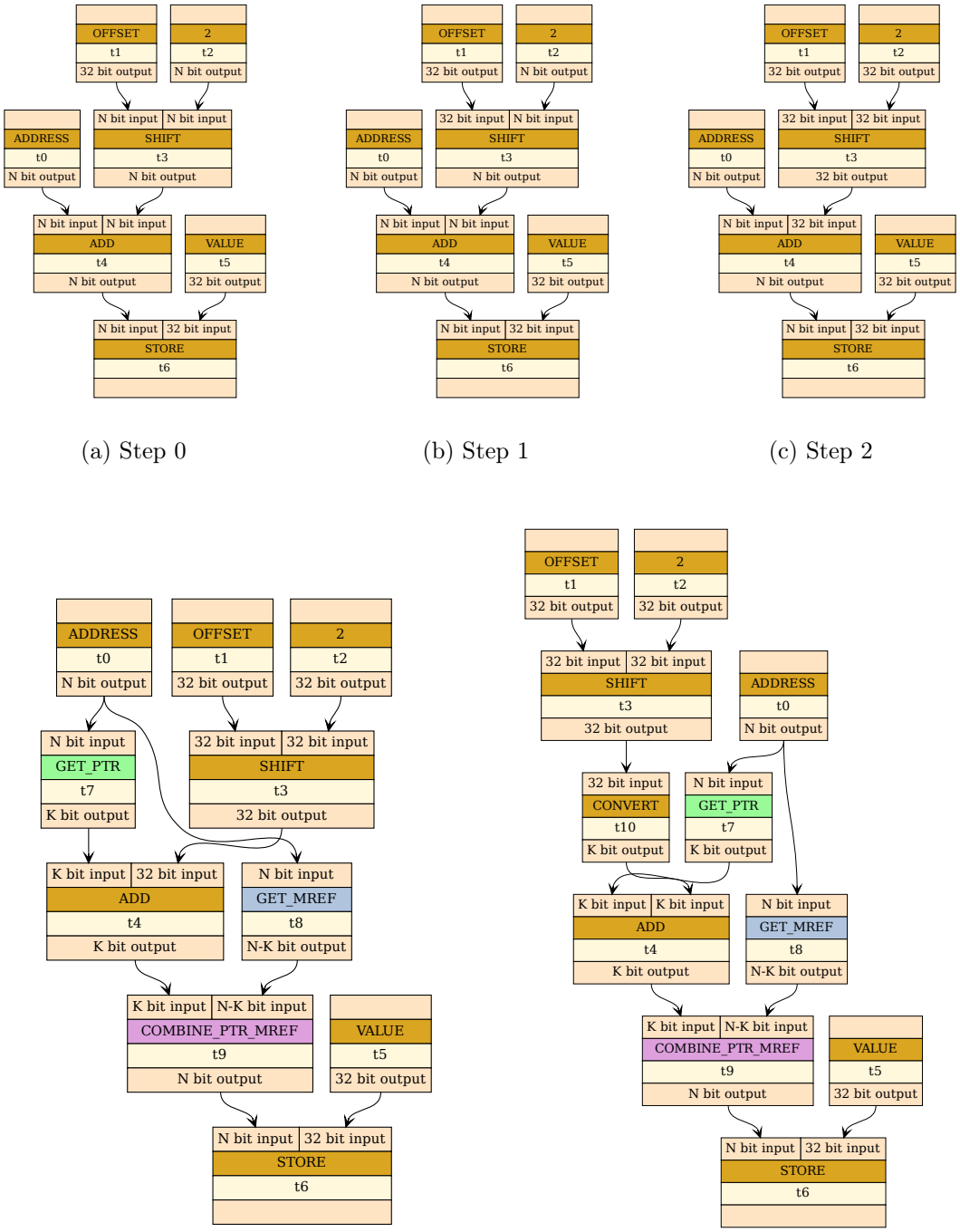
node, t4 needs both its inputs to have the same size. Figure 7e shows inserting t10, a CONVERT node. If K is greater than 32, this would be a sign extension. If K is less than 32, this would be a truncation. If K is actually 32, then this would not need to be inserted at all.

This is a fairly simple example. Yet many steps are required to create a valid DAG that can be selected to instructions. This can impact optimization passes that require specific orderings of instructions.

4.4 Usability

The compiler should have no sharp edges. This requires good system design to make it user friendly. The compiler should also be predictable and do what is expected. Take a C compiler for a capability architecture. Most developers writing C code already understand the semantics of C, it is not in the best interest of the compiler to change these semantics drastically. This creates two problems. First, the developer now has a steeper learning curve to learn a new set of programming semantics. Second, now there is something that looks like C, is written like C, but does not have C semantics. This can be incredibly confusing and is likely to cause a developer to misunderstand the semantics they are writing. This means that whatever semantics are changed must be related to things that are transparent to the programmer, such as the semantics of a metadata reference (this is discussed in the next section). We note that this particular topic has been previously studied [36].

An important part of this is making sure the compiler generates human friendly assembly code while still being optimized. This principle guided the development of the above presented ideas. The less drastic and confusing changes made to the outputed



(a) Step 0 (b) Step 1 (c) Step 2

(d) Step 3 (e) Step 4

Figure 7. Storing to an Array

assembly, the easier for a developer to debug that code. This is heavily dependant upon how much optimization is done to the code. A developer should still be able to disable optimizations and reason about the generated code in comparison to the source. An intrepid developer should be able to analyze the optimized generated code and investigate why the compiler chose this sequence of code to be optimal. This creates an environment where developers actively want to work on a platform. Providing a rich development environment that is developer friendly encourages adoption. Developers need less encouragement to a adopt a new system if it is familiar. This takes on almost a sales-like approach, but a stubborn programmer needs persuading.

Part of this usability can actually be baked into the language. An architecture like Beta with transparent registers does not need to expose this directly to a source language. That is what assembly is for. But if it is exposed to the source in a clean and usable way, it lowers the barrier to entry to advanced use cases. Inline assembly allows for direct access to the instruction set, but can be messy and painful to use. This is the motivating case for builtins. Compiler builtins, also called intrinsics, are functions that can be called from source code that map to a very specific set of assembly. For every DAG operation defined, the equivalent builtin function can be exposed to the source language. This essentially allows a developer to write DAG mapping themselves, either where they don't trust the compiler to do it right or they are writing some pattern not natively supported by the compiler. An example of this is show in the next section in Code Snippet 8.

4.5 Pointer Semantics

Just as important to the discussion of actually compiling code is the semantics of writing such code. For example, the value of 0 is commonly used as a null pointer, describing an invalid address. There are also a specific set of operations in a high language like C that can be done on pointers. Note that this is regardless of hardware limitations. On a non-capability architecture, there is no semantic difference given to a pointer verses a raw integer. Of course it is entirely possible to do these operations from a high level language by converting the type from a pointer to an integer. Then any integer operation is valid. This is called *type punning*.

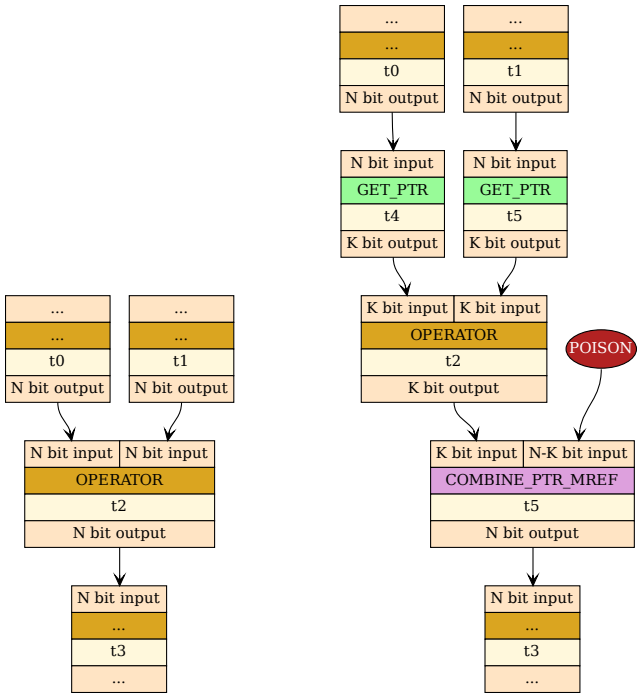
Temporarily ignoring type punning, there is only one common pointer arithmetic operations where both operands are pointers, subtraction Note this is not a discussion of a pointer minus an integer, but a pointer plus or minus another pointer. A pointer difference can be used to know the size of an array given a start and end pointer. But the result is not a valid pointer, it is an integer. The compiler can safely ignore the immutable metadata reference in this case. Either the compiler can insert some other value as the resulting immutable metadata reference or it can pick one of the two operands to preserve. Regardless, the hardware will fault if the result is attempted to be used as a capability.

Considering type punning, the casting of a capability to an integer is a tantamount to extracting the memory address to a general purpose register. This allows a high level language to arbitrarily modify the memory address. To be usable as a memory address, it must be casted back to a capability. This is a loss of information, and the compiler must copy the metadata reference to the result to form a valid capability. But if multiple capabilities are casted to integers and used to compute a single result,

it is unclear which metadata reference should be copied to the result. A logical choice is to just use the first one. Our generic heuristic takes a different approach.

We update our mapping heuristics with these semantics. As an example, consider the pointer difference case ($pointer2 - pointer1$). The simple thing, especially in the case where there is no result, is to just operate on the K bit part. Given a typical three-address instruction, with two operands and one result with both operands as valid full length N bit operands, e.g. not constants. This is show in Figure 8a. This is wrapped trivially in `GET_PTR` for the inputs. For any such operation where there are no outputs, the problem is solved, as no new N bit result is created. Metadata references are not visible to the programmer, and so it does not make sense to make this a usable operation. This also falls in line with the C standard [18]. This is represented this in our mapping with a symbolic `POISON` value, seen in Figure 8b. This creates a valid pointer and an invalid metadata reference. The `POISON` representation generates no additional code, it is a compile concept only. This signifies that we no longer care about the value of the metadata reference, from the point of view of the compiler it is invalid. Its real value is dependant upon register allocation and other optimizations that occur later. This follows the C standard for the pointer difference case and we generalize it to any such operation on two pointers. In any such case, the operation will create an invalid metadata reference.

Note that similar work has been presented on these pointer semantics. The CHERI architecture has similar problems within their compiler [36]. However much of their work focused on defining how to fix those problems in software. One of the most obvious examples is the casting of a `long` to a `intptr_t`, in which an invalid capability is created [23]. A compiler warning is emitted for these cases, but it is not specified



(a) Original (b) Poison inserted

Figure 8. Two Operand Case

```
1 void* align(void* p, unsigned n) {
2     unsigned b = (1 << n) - 1;
3
4     intptr_t p_ = (intptr_t)p;
5     p_ = (p_ + b) & ~b;
6     p = (void*)p_;
7
8     return p;
9 }
```

Code Snippet 7. Function to Align a Pointer

```
1 void* align(void* p, unsigned n) {
2     unsigned b = (1 << n) - 1;
3     mref_t m = get_mref(p);
4     intptr_t p_ = (intptr_t)p;
5     p_ = (p_ + b) & ~b;
6     p = (void*)p_;
7     p = set_mref(p, m);
8     return p;
9 }
```

Code Snippet 8. Revised Code Snippet 7 for Capabilities

what can compiler do in these cases in terms of optimization. From a separable capability pointer point of view, if a user casts a `long` to a `intptr_t`, a warning should be emitted as this is potentially a mistake on the part of the user. However, given the code as-is, the compiler can take the approach frequently used with undefined behavior. The user is expressing the intent, whether on purpose or not, that they want a memory address. So the compiler can produce the correct code to create a memory address, but since there is no metadata reference to associate it is free to remove the code that might have maintained the integrity of the capability. This would be a correct compiler optimization.

This heuristic breaks under certain common code patterns. Consider the code in Code Snippet 7, this function will align a pointer to a certain number of bits. Calling this function with $nbits = 12$ returns a $4KiB$ aligned pointer. This is a valid operation, particularly used in high performance code, to ensure proper alignment that matches hardware requirements. But under the outlined poison semantics, information may be lost by the compiler. The casting of the pointer between an integer and a capability is likely to produce incorrect code. Using the builtins previously discussed it is possible to rewrite this code to be fully correct even in the face of compiler optimizations, as show in Code Snippet 8.

Chapter 5

COMPILING FOR ZENO

A subset of the previously described techniques have been used to build a compiler for the Zeno architecture. The techniques were developed first for the Zeno compiler, then expanded and generalized for a generic architecture, and then reapplied to the Zeno compiler. RISC-V/Zeno introduces many of the same complexities as the Alpha/Beta system that the theoretical techniques are for. It is important to note that only a subset of theoretical ideas are implemented in Zeno. The DAG mapping approach is designed to be incredibly flexible, but the Zeno compiler only needs a few heuristics to be workable. The pseudo approach is nearly identically used in the Zeno compiler to do instruction selection and to implement many higher level machine instructions in LLVM.

5.1 Zeno ISA

The Zeno architecture builds on top of the 64 bit RISC-V specification [38, 33]. The standard RISC-V I instruction set, as well as the M and A extensions are implemented. On top of this is added a subset of the xBGAS extended global addressing extension [21]. Zeno uses the RISC-V 64 bit general purpose registers as its memory addressing registers and the mirrored xBGAS 64 bit extended registers as its metadata reference register. Zeno Namespaces, an abstraction for a region of memory, serve as the the capabilities for this architecture. Namespace Identifiers (NSIDs) are the concretization of a metadata reference. These registers are summarized in Figure 9. Note that the

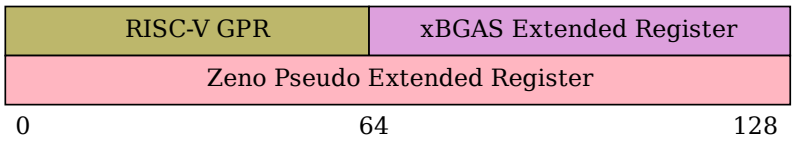


Figure 9. Registers for Zeno

Pseudo Extended Register is not a real register on Zeno, this is a compiler construct used to generate valid Zeno assembly code.

All RISC-V I, M, and A instructions are implemented. On top of this are a few xBGAS instructions. We will enumerate the additions to the base RISC-V ISA from the *software* view, not the microarchitectural view. Table 1 lists the added instructions. All instructions listed also have an explicit pseudo version. The explicit pseudo version of instruction `XYZ` is `PseudoXYZ`. Some of the pseudo versions have an additional immediate argument. This is used to aid the register allocation process. Those instructions marked as explicit pseudo instructions are not real instructions, they are intrinsically pseudo instructions and may have an additional explicit Pseudo form.

Zeno machine instructions are encoded in LLVM using TableGen. (See Zeno instruction encoding in Table 1). There are three classes of instructions: loads, stores, and address management. The loads are the Beta `LOAD` instruction, the stores are the Beta `STORE`, and the address management are the Beta `COPY`. The different variations of loads and stores correspond to the RISC-V base loads and stores, `ELW` relates to `LW`. These are all fundamentally the same in how they use the pointers, they each take a pointer operand as `rs1`. `rs1` specifies both the general purpose register and the extended register. In their explicit pseudo form, the loads and stores take an extra immediate argument. This encodes an optional register selection. If left as

Table 1. Zeno Instructions on RISC-V

Instruction	**Beta Equivalent**	**Destination Register(s)**	**Source Register(s)**	**Pseudo?**	**Extra Pseudo Argument?**
ELD	LOAD	GPR rd	GPR rs1 EXT rs1	No	Yes
ELP	LOAD	GPR rd EXT rd	GPR rs1 EXT rs1	Yes	Yes
ELW	LOAD	GPR rd	GPR rs1 EXT rs1	No	Yes
ELH	LOAD	GPR rd	GPR rs1 EXT rs1	No	Yes
ELHU	LOAD	GPR rd	GPR rs1 EXT rs1	No	Yes
ELB	LOAD	GPR rd	GPR rs1 EXT rs1	No	Yes
ELBU	LOAD	GPR rd	GPR rs1 EXT rs1	No	Yes
ELE	LOAD	EXT rd	GPR rs1 EXT rs1	No	Yes
ESD	STORE		GPR rs1 EXT rs1 GPR rs2	No	Yes
ESP	STORE		GPR rs1 EXT rs1 GPR rs2 EXT rs2	Yes	Yes
ESW	STORE		GPR rs1 EXT rs1 GPR rs2	No	Yes
ESH	STORE		GPR rs1 EXT rs1 GPR rs2	No	Yes
ESB	STORE		GPR rs1 EXT rs1 GPR rs2	No	Yes
ESE	STORE		GPR rs1 EXT rs1 EXT rs2	No	Yes
EADDI	COPY	GPR rd	EXT rs1	No	No
EADDIE	COPY	EXT rd	GPR rs1	No	No
EADDIX	COPY	EXT rd	EXT rs1	No	No

-1, the register allocator will select the most appropriate physical register. If set to a number in the range $0-31$, these instructions are preset to the corresponding physical register before the register allocator runs. This allow the implementation to choose physical registers explicitly in early stages of the compiler pipeline, before it is normally possible to select a physical register. Code Snippet 9 demonstrates how all of these options can be encoded in a single pattern. This pattern can be instantiated to generate many different instruction definitions, and automatically generate associated tools such as an assembler and dissembler.

```
multiclass RV64LoadExtended<
    bits<3> funct3,
    string opcodestr,
    string schedSize,
    string loadRegClass = "GPR",
    bit hasConstraint = 0> {
    let hasSideEffects = 0,
     mayLoad = 1,
     mayStore = 0 in {
      def ""#NAME
        : RVInstI<
          funct3,
          OPC_ELOAD,
          (outs !cast<RegisterClass>(""#loadRegClass):$rd),
          (ins GPR:$rs1, simm12:$imm12),
          opcodestr,
          "$rd, ${imm12}(${rs1})"
        >,
        Sched<[!cast<SchedWrite>("WriteLD"#schedSize), ReadMemBase]>;
      let hasPostISelHook = 1,
       isAsmParserOnly = 0,
       isCodeGenOnly = 0,
       isPseudo = 1 in {
        def Pseudo#NAME
          : Pseudo<
            (outs !cast<RegisterClass>(""#loadRegClass):$rd),
            (ins PXER:$rs1, simm12:$imm12, simm12:$regID),
            [],
            opcodestr#"_p",
            "$rd, ${imm12}(${rs1}), $regID"> {
              let Constraints =
                !if(hasConstraint, "@earlyclobber $rd", "");
            }
      }
    }
    def
      : InstAlias<
          opcodestr#" $rd, (${rs1})",
          (!cast<RVInstI>(""#NAME)
            !cast<RegisterClass>(""#loadRegClass):$rd,
```

```
41              GPR:$rs1,
42              0
43            ),
44          0>;
45 }
46 defm ELD  : RV64LoadExtended<0b011, "eld",  "D">;
47 defm ELP  : RV64LoadExtended<0b011, "eld",  "D", "PXER", 1>;
48 defm ELW  : RV64LoadExtended<0b010, "elw",  "W">;
49 defm ELH  : RV64LoadExtended<0b001, "elh",  "H">;
50 defm ELHU : RV64LoadExtended<0b101, "elhu", "H">;
51 defm ELB  : RV64LoadExtended<0b000, "elb",  "B">;
52 defm ELBU : RV64LoadExtended<0b100, "elbu", "B">;
53 defm ELE  : RV64LoadExtended<0b111, "ele",  "D", "ER64">;
```

Code Snippet 9. Pattern for Generating Load Instruction Definitions in TableGen

There are two special instructions, `ELP` and `ESP`. These are true pseudo instructions and they have an explicit pseudo form as well with the register selection argument. These are explicit 128 bit memory operations that correspond to a general purpose register 64 bit load/store and an extended register 64 bit load/store. The 'pointer' memory operations simplify the instruction selection and optimization process by using the pseudo 128 bit registers.

An important component of the Zeno compiler is the overloading of operations. Any RISC-V instruction that normally takes a general purpose register can also take a pseudo extended register. This makes the instruction into a pseudo instruction. At the final stage of the machine instruction phase of the compiler, any instruction using pseudo registers is lowered to the appropriate general purpose register and a register copy for the extended register is inserted.

5.2 DAG Mapping

The Zeno compiler implements nearly all of the DAG mapping operators, only `SPLIT_PTR_MREF` is not implemented. Since a Zeno metadata reference is a NSID,

in the implementation `MREF` is renamed to NSID. This gives the following five DAG mapping operations.

- `ZENO_GET_PTR`
- `ZENO_SET_PTR`
- `ZENO_GET_NSID`
- `ZENO_SET_NSID`
- `ZENO_COMBINE_PTR_NSID`

The common full capability wrapping is applied in all cases, with the exception of implementing the builtins. This only requires `ZENO_GET_PTR`, `ZENO_GET_NSID`, and `ZENO_COMBINE_PTR_NSID`. `ZENO_GET_NSID` and `ZENO_SET_NSID` are both used to implement builtins. Although it is implemented, `ZENO_SET_PTR` is actually unused. Since Zeno memory addresses are 64 bits and NSIDs are 64 bits, we define the DAG operator bit width N to be 128 and K to be 64. The rest of the heuristics and ideas of DAG mapping remain largely unchanged. We do note that the heuristic for handling an operator with a capability result and more than one capability operand is only partially implemented. This was implemented simply to handle a branching statement that compares pointers. Other multi-operand pointer arithmetic operations are supported, but not explicitly handled as previously described with poison values.

Figure 10 provides an overview of the entire SelectionDAG pipeline. For context, all of the blue sections are existing components of SelectionDAG. There exists large elements of this that are not shown. The green sections are the added parts of the pipeline for DAG mapping. Note that a green section inside of a blue means that an existing component was augmented with DAG mapping, rather than constructed from scratch. The purple and red sections are a logically grouping of the DAG mapping actions performed.

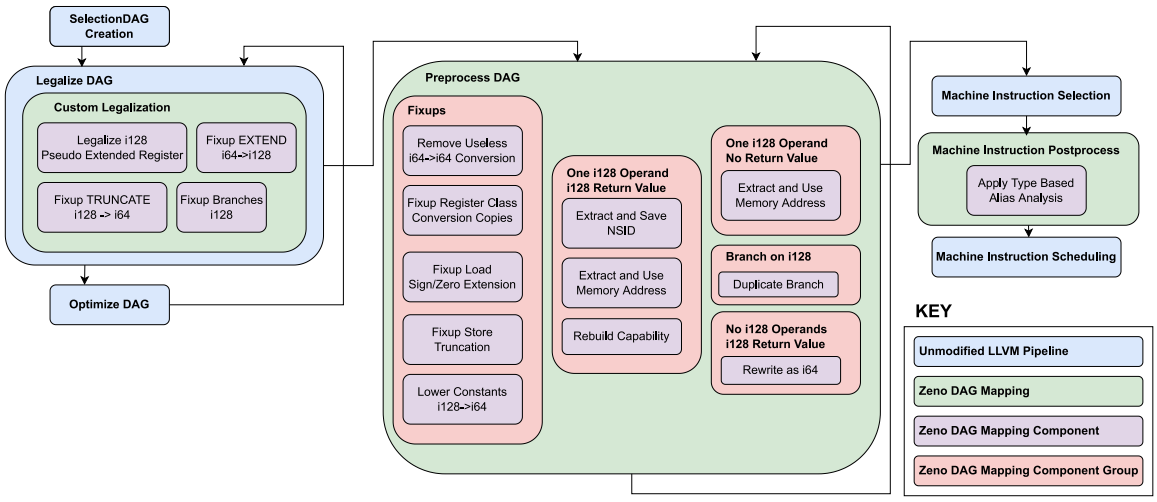
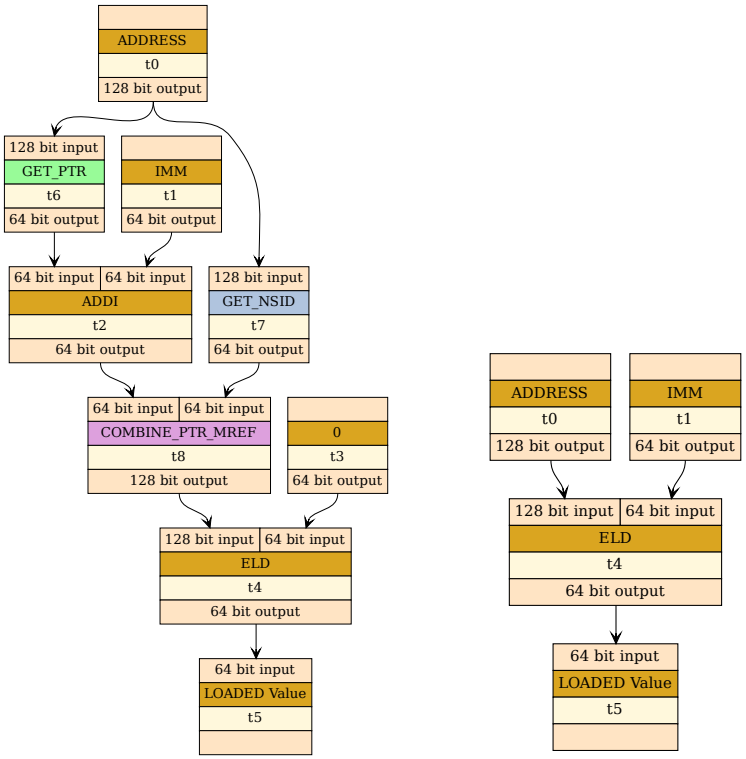


Figure 10. SelectionDAG Pipeline for Zeno

Note that the majority of DAG mapping does not modify the existing optimization phase or instruction selection and scheduling. The DAG mapping is added as pre/post processing. In essence, making the DAG legal and valid for Zeno in terms that the existing LLVM pipeline can understand. This was an intentional design choice in line with the core ideas of DAG mapping. The less that the existing LLVM compiler is modified, the more portable and reusable the DAG mapping is. This allows the Zeno compiler to easily add the DAG mapping on top of existing components and reuse all of the optimization goodness that already is built in to the LLVM compiler.

In this context, DAG mapping is essentially a way to legalize types for a capability architecture. The additional machine instructions that are emitted are purely register to register copies. This is highly advantageous, as passes already exist to optimize register to register copies. Dead code elimination passes and register coalescing passes can run largely unmodified. Assisted by Zeno specific optimization passes, many of these copies can be eliminated.

One key component of the RISC-V instruction set is the ability to encode constant



(a) ADDI + ELD with DAG Operators (b) Optimized ELD

Figure 11. Optimization Example of ELD

offsets directly in a memory operation. For example, `ld rd, imm(rs1)` will access the memory at address $GPR[rs1] + imm$. There exist optimization passes in the compiler to fold a constant immediate into memory accesses like this for RISC-V. However, DAG mapping for Zeno breaks this optimization pass. Consider the following DAG in Figure 11a. The additional DAG operators are required to make a valid DAG, but the optimization pass for folding constants will be confused by the additional passes. This example is the reason only full capability preservation is implemented in the Zeno compiler, cases like this are easy patterns to spot and then optimize. After applying the matching pattern to this DAG, the desired DAG with folded constants is produced Figure 11b.

5.3 Machine Instruction Passes

There are a number of specialized passes for Zeno that operate on Machine Instructions, see Table 2. These passes can be classified as optimization passes, legalization passes, and code generation passes. Optimization passes for the most part are restricted to dead code removal, specifically dead register copies. A dead register copy is where a value is copied to a destination register, and then the value in the destination register is never used again. Many of these copies get generated by the DAG mapping, so these passes are critical. The legalization passes perform all kinds of fix-ups on instructions, at all stages of the machine instruction passes. For example, a pass runs shortly after machine instruction selection and fixes invalid instructions that are generated. Code generation passes convert pseudo instructions, including real instructions with pseudo operands, to real instructions.

One of the most important parts of this is propagating metadata references. Any pseudo operands remaining are assumed to be real capabilities that need to be preserved. Therefore part of the conversion to real operands is the insertion of register copies to preserve the NSID. This typically occurs under pointer arithmetic, where the destination operand is a capability and the first operand is also a capability. The correct transformation is to lower the pseudo capability registers to real integer registers and insert an associated copy instruction for the real extended registers.

5.4 Builtins and Intrinsics

Implemented in the compiler are two Namespace management builtins and two explicit load/store builtins that are overloaded. These builtins are implemented in

Table 2. Zeno Machine Instruction Passes

Pass	**File**	**Classification**
Resolve Physical Registers To Pseudo Registers For ELP/ESP	RISCVZenoDAGFixup.cpp	Legalization
Remove Useless Register Copy With SRC=DST	RISCVZenoDeadCode.cpp	Optimization
Remove Adjacent Duplicate Extended Register Copy	RISCVZenoDeadCode.cpp	Optimization
Remove Copies Of Registers That Are Never Used	RISCVZenoEliminateExtraCopies.cpp	Optimization
Remove Empty Full Capability Preservation Mapping	RISCVZenoEliminateExtraCopies.cpp	Optimization
Remove Double Copies	RISCVZenoEliminateExtraCopies.cpp	Optimization
Expand Explicit Pseudo Instructions To Real Instructions	RISCVZenoExpandPseudo.cpp	Code Generation
Special Expansion Of The Stack Frame NSID	RISCVZenoPreEmitFixup.cpp	Code Generation
Expand Real Instructions with Pseudo Operands to Real Instructions	RISCVZenoPreEmitFixup.cpp	Code Generation
Clean Up Remaining DAG Operators	RISCVZenoPreEmitFixup.cpp	Legalization Code Generation
Select Explicit Physical Registers	RISCVZenoSelectExtReg.cpp	Legalization

the C and C++ front end of LLVM, and map directly to LLVM IR intrinsics. The IR intrinsic are language agnostic.

The two Namespace management builtins, Table 3, allow a programmer to directly manipulate the NSID stored in the extended register file. Using these two builtins, it is possible for programmers to build capabilities entirely from C/C++ without resorting to inline assembly. This is a huge quality of life improvement for developers writing trusted firmware or an OS for Zeno.

The two explicit load/store builtins are heavily overloaded, see Table 4, Table 5, Table 6, and Table 7. The builtin name is the same and the correct LLVM IR intrinsic is inferred from the type of the arguments. There are also two versions of each builtin. The first version just takes a pointer (and a value, in the case of the stores) and executes the corresponding instruction(s). This is identical to if the programmer had just use the pointer in a normal way in a C/C++ program. The second version takes the same arguments as the first, plus an additional parameter to be used as the NSID. This allows a programmer to use an arbitrary NSID for a memory access, without modifying the pointer explicitly. This provides additional flexibility to the programmer.

These builtins break the transparency of the compiler. Previously a programmer was entirely agnostic of Zeno, these builtins are explicit Zeno operations. This breaking of transparency provides a way for programmers to have more control over the code generated. This additional control essentially comes for free, the builtins serve no detriment to the compiler. If a programmer didn't know these builtins existed and never used them, they could still write and compile correct programs. If the programmer does use the builtins, we are making the assumption that they know what they are doing. Any bugs they might introduce by using the builtins improperly

Table 3. Intrinsics for NS Management

C/C++ Builtin prefixed by __builtin_riscv_zeno_	**Argument Type(s)**	**Return Type**	**LLVM Intrinsic** prefixed by llvm.riscv.zeno.	**Instruction**
set_nsid	void* unsigned long	void*	set.nsid	EADDIE
get_nsid	void*	unsigned long	get.nsid	EADDI

Table 4. Intrinsics for __builtin_riscv_zeno_load with 8 and 16 Bit Values

Argument Type(s)	**Return Type**	**LLVM Intrinsic**	**Instruction**
char*	char	llvm.riscv.zeno.load.u8	ELBU
unsigned char*	unsigned char		
char* unsigned long	char	llvm.riscv.zeno.load.u8.nsid	EADDIE ELBU
unsigned char* unsigned long	unsigned char		
signed char*	signed char	llvm.riscv.zeno.load.s8	ELB
signed char* unsigned long	signed char	llvm.riscv.zeno.load.s8.nsid	EADDIE ELB
unsigned short*	unsigned short	llvm.riscv.zeno.load.u16	ELHU
unsigned short* unsigned long	unsigned short	llvm.riscv.zeno.load.u16.nsid	EADDIE ELHU
short*	short	llvm.riscv.zeno.load.s16	ELH
signed short*	signed short		
short* unsigned long	short	llvm.riscv.zeno.load.s16.nsid	EADDIE ELH
signed short* unsigned long	signed short		

is introduced by the programmer, so it is on them to fix it. The builtins also do not break any security guarantees made by the Zeno architecture. The hardware enforces that only trusted firmware can create Namespaces and software that uses builtins attempting to do so will trap.

Table 5. Intrinsics for __builtin_riscv_zeno_load with 32, 64, and 128 Bit Values

Argument Type(s)	**Return Type**	**LLVM Intrinsic**	**Instruction**
int*	int	llvm.riscv.zeno.load.32	ELW
signed int*	signed int		
unsigned int*	unsigned int		
int* unsigned long	int	llvm.riscv.zeno.load.32.nsid	EADDIE ELW
signed int* unsigned long	signed int		
unsigned int* unsigned long	unsigned int		
long*	long	llvm.riscv.zeno.load.64	ELD
signed long*	signed long		
unsigned long*	unsigned long		
long* unsigned long	long	llvm.riscv.zeno.load.64.nsid	EADDIE ELD
signed long* unsigned long	signed long		
unsigned long* unsigned long	unsigned long		
void**	void*	llvm.riscv.zeno.load.128	ELD ELE
void** unsigned long	void*	llvm.riscv.zeno.load.128.nsid	EADDIE ELD ELE

5.5 Evaluation

The Zeno compiler is evaluated in a number of ways to ensure both correctness and to get an understanding of the performance impacts. The full impacts will not be fully realized until the Zeno platform reaches more stability and more benchmarks can be validated on it. It is possible to get a theoretical idea of some of the potential performance pitfalls, and an insight into future optimizations to solve them before they ever become an issue.

Table 6. Intrinsics for __builtin_riscv_zeno_store with 8 and 16 Bit Values

Argument Type(s)	**LLVM Intrinsic**	**Instruction**
char* char signed char* signed char unsigned char* unsigned char	llvm.riscv.zeno.store.8	ESB
char* unsigned long char signed char* unsigned long signed char unsigned char* unsigned long unsigned char	llvm.riscv.zeno.store.8.nsid	EADDIE ESB
short* short signed short* signed short unsigned short* unsigned short	llvm.riscv.zeno.store.16	ESH
short* unsigned long short signed short* unsigned long signed short unsigned short* unsigned long unsigned short	llvm.riscv.zeno.store.16.nsid	EADDIE ESH

Table 7. Intrinsics for __builtin_riscv_zeno_store with 32, 64, and 128 Bit Values

Argument Type(s)	**LLVM Intrinsic**	**Instruction**
int* int	llvm.riscv.zeno.store.32	ESW
signed int* signed int		
unsigned int* unsigned int		
int* unsigned long int	llvm.riscv.zeno.store.32.nsid	EADDIE ESW
signed int* unsigned long signed int		
unsigned int* unsigned long unsigned int		
long* long	llvm.riscv.zeno.store.64	ESD
signed long* signed long		
unsigned long* unsigned long		
long* unsigned long long	llvm.riscv.zeno.store.64.nsid	EADDIE ESD
signed long* unsigned long signed long		
unsigned long* unsigned long unsigned long		
void** void*	llvm.riscv.zeno.store.128	ESD ESE
void** unsigned long void*	llvm.riscv.zeno.store.128.nsid	EADDIE ESD ESE

5.5.1 Static Tests

Zeno has a suite of static tests to validate the code generation. These are a set of small, exemplar programs that are intentionally written to produce certain effects. It is specifically written to stress the DAG mapping and associated passes. The suite is written in C, rather than the normal LLVM tests for a backend which are written in LLVM IR[24]. This is because many of the bugs that occur are due to incorrect LLVM assumptions when converting C to LLVM IR, and frankly because C is much easier to read and reason about than LLVM IR.

The static tests are constructed in the following sequence.

1. Construct minimal C sample to produce desired effect
2. Hand assembly a solution
3. Run the compiler and compare results
4. Make small, "insignificant" tweaks
5. Repeat steps 2 through 4 for various interesting combinations of compiler flags.

This constructs the actual tests, a test harness can then automatically run the tests during development and ensure that no test gets broken. Note, an "insignificant" tweak is just little spacing and ordering changes that do not effect the overral result. For example, consider adding `a0`, `a1`, and `a2`, storing the result in `a0`. Code Snippet 10 shows the different perfectly valid permutations of instructions, which are all equivalent and just depend on the scheduling and register selection. This have no effect on the actual test case.

The test suite currently contains 198 individual tests and many more exist but have yet to be defined. All but one test pass and we will deep dive on it. Additionally,

```
1  ;; Possibility 1
2  add a0, a1, a0
3  add a0, a0, a2
4  ;; Possibility 2
5  add a0, a1, a0
6  add a0, a2, a0
7  ;; Possibility 3
8  add a0, a2, a0
9  add a0, a0, a1
10 ;; and so on...
```

Code Snippet 10. Possible Permutations for Adding Three Registers

rather than showing all the test cases that work we will select a handful to analyze of the ones that are functionally correct, but have a few noteworthy characteristic or future optimization opportunities.

Code Snippet 11 is a test case that stores a value to a 2D array (a double pointer). This should be a simple case; load the pointer at an offset of 30, use the loaded pointer to store a value at an offset of 60. The Zeno compiled version, Code Snippet 12, correctly performs the memory accesses but a closer examination shows an inefficiency. Lines 4 and 5 make a copy of the loaded pointer before storing a value to it. This is correct code, but we do not need the extra copies as line 6 will not overwrite the pointer. Furthermore even if the pointer gets partially overwritten we no longer care, as the pointer is not used for the rest of the function. A better version would look like Code Snippet 13, which eliminates the register copy. This test case demonstrates that an improved register coalescing optimization is needed prior to physical register allocation.

Note that the opportunity for optimization in Code Snippet 11 is likely from the result of DAG mapping. Code Snippet 14 is a similar code sample, but instead of fixed array offsets known at compile time, the offsets are parameters to the function; they are unknown at compile time. Therefore the compiler cannot fold them into

```
1 void func(int** p, int v) {
2   p[30][60] = v;
3 }
```

Code Snippet 11. Storing a Value to a 2D Array with Fixed Offsets

```
1 func:
2   eld a2, 480(a0)
3   ele e12, 488(a0)
4   mv a0, a2
5   moveee e10, e12
6   esw a1, 240(a0)
7   ret
```

Code Snippet 12. Result of Compiling Code Snippet 11 with -O3

```
1 func:
2   eld a2, 480(a0)
3   ele e12, 488(a0)
4
5
6   esw a1, 240(a2)
7   ret
```

Code Snippet 13. Optimal Version of Code Snippet 12

the memory operations, so the compiler must generate the instructions to do pointer arithmetic. Note the register copies resulting from DAG mapping on lines 3 and 5, and then again on lines 9 and 11 in Code Snippet 15. In both cases, there is essentially a duplicate copy. This is caused not by DAG mapping, but by an overlap between DAG mapping and pseudo lowering. The pseudo instructions are meant to be compilation targets for the DAG mapping as an intermediate step and they are lowered to the proper instruction(s). The code shown is not being properly lowered, and so these duplicate copies are the result of later passes in the compiler making a last ditch effort to remove all pseudo instructions. So they are functioning correct, as no pseudo operations should be emitted. But the pseudo instructions should have already been lowered.

This is only half the problem, as all four lines are not needed. Code Snippet 16 shows what the code should ultimately be compiled to. The root cause here is again useless register copies, they make the code functionally correct but a good optimizer should remove them. This example can also be solved by an improved register coalescing pass.

```
1 void func(int** p, int offset1, int offset2, int v) {
2   p[offset1][offset2] = v;
3 }
```

Code Snippet 14. Storing a Value to a 2D Array with Variable Offsets

```
1  func:
2    slli a1, a1, 4
3    moveee e11, e10
4    add a1, a0, a1
5    mv a0, a1
6    eld a1, 0(a0)
7    ele e11, 8(a0)
8    slli a0, a2, 2
9    moveee e10, e11
10   add a0, a1, a0
11   moveee e10, e11
12   esw a3, 0(a0)
13   ret
```

Code Snippet 15. Result of Compiling Code Snippet 14 with -O3

```
1  func:
2    slli a1, a1, 4
3
4    add a0, a0, a1
5
6    eld a1, 0(a0)
7    ele e11, 8(a0)
8    slli a2, a2, 2
9
10   add a1, a1, a2
11
12   esw a3, 0(a0)
13   ret
```

Code Snippet 16. Optimal Version of Code Snippet 15

Code Snippet 17 is a simpler case of Code Snippet 14 as it is just a single pointer that a value is being stored to. In Code Snippet 18 pseudo instructions are again not removed early enough in the pipeline. The compiler takes the conservative approach and leaves in the pseudo operations and are lowered crudely to real instructions later, albeit correctly. Furthermore, these instructions are again not needed as this example could be optimally compiled with an improved register coalescer as Code Snippet 19. What we find is that the majority of simple pointer arithmetic cases like this have this inefficiency. The code is functionally correct and will run properly on hardware, but the optimizations are too late and not aggressive enough to achieve the best possible result.

Code Snippet 20 and Code Snippet 22 showcase one of the more perplexing optimization opportunities. They are both functions that take a pointer to a struct

```
1 void func(unsigned char* p, int offset, unsigned char v) {
2   p[offset] = v;
3 }
```

Code Snippet 17. Storing a Value at a Variable Offset

```
1 func:
2   moveee e11, e10
3   add a1, a0, a1
4   mv a0, a1
5   esb a2, 0(a0)
6   ret
```

Code Snippet 18. Result of Compiling Code Snippet 17 with -O3

```
1 func:
2
3   add a0, a0, a1
4
5   esb a2, 0(a0)
6   ret
```

Code Snippet 19. Optimal Version of Code Snippet 18

containing two elements and the function returns the sum of the two elements. If an element of the struct is a pointer, the pointer is dereferenced and that is the value that is summed. Code Snippet 20 is argulablty the more complex case as it uses a struct that has two pointers, but the compiler produces the correct assembly (Code Snippet 21). This is not the case with Code Snippet 22, where the compiler produces Code Snippet 23. This is the same extra register copy as Code Snippet 11 and could theoretically be solved the same way to produce the desired output in Code Snippet 24. But what this test case shows is that there may be something else at work that is not fully understood yet. The more complex test case successfully passes through the existing optimization passes and gets the correct result. This is a very important puzzle, because it actually holds the answer to solve the optimization problem. This will eventually be solved by comparing what is different in the code generation pipeline between the working test case and the broken test case and then using that information to improve the code generation.

Code Snippet 25 is only currently failing test case for functionality. It involves

```
struct s { int* x; int* y; };
int func(struct s* s) {
  return *s->x + *s->y;
}
```

Code Snippet 20. Using a Struct Containing Two Pointers

```
func:
  eld a1, 0(a0)
  ele e11, 8(a0)
  eld a2, 16(a0)
  ele e12, 24(a0)
  elw a0, 0(a1)
  elw a1, 0(a2)
  addw a0, a1, a0
  ret
```

Code Snippet 21. Result of Compiling Code Snippet 20 with -O3

```
struct s { int x; int* y; };
int func(struct s* s) {
  return s->x + *s->y;
}
```

Code Snippet 22. Using a Struct Containing One Pointer and One Integer

```
func:
  mv a1, a0
  moveee e11, e10
  eld a2, 16(a1)
  ele e12, 24(a1)
  elw a0, 0(a0)
  elw a1, 0(a2)
  addw a0, a1, a0
  ret
```

Code Snippet 23. Result of Compiling Code Snippet 22 with -O3

```
func:


  eld a1, 16(a0)
  ele e11, 24(a0)
  elw a0, 0(a0)
  elw a1, 0(a1)
  addw a0, a1, a0
  ret
```

Code Snippet 24. Optimal Version of Code Snippet 23

a function passing a pointer to a function that takes an integer. From the point of view of the compiler, since it doesn't know the contents of the function being called, the argument is an integer. But the programmer knows that the integer argument is actually going to be used as a memory address. This is a minimal example of real world code that does similar operations such as this, such as inside of the Linux kernel. So this is an important test case to get right; and the compiler does get it correct in Code Snippet 27, the one caveat being that this is unoptimized code. Even though the argument is an integer, the compiler stills sets up the argument with the NSID for the pointer because that is the cautious and functional thing to do. When optimizations are applied in Code Snippet 26, the test case breaks. Now instead of the NSID being copied properly, those instructions are entirely optimized away.

This unfortunately is a case of the optimization passes being too aggressive. In all of the previous test cases examined, we concluded a more aggressive optimizer was needed, but in this last case the optimizer is too aggressive. Unfortunately, the less aggressive approach is the one that has to be taken. Because even though the breaking test case is an edge case and represents a small portion of all software, the compiler must always produce functionally correct code no matter the input. Optimization is a bonus, not a requirement.

5.5.2 File Size

We compiled *newlib*, a bare metal C standard library implementation, using both a normal LLVM build for RISC-V IMA and the Zeno compiler. Newlib was compiled with $-Os$, meaning we optimized for size compared to just purely speed. This will

```
1 void unknown(long ptr, long v);
2 void func(long v, long* ptr) {
3   unknown((long)ptr, v);
4 }
```

Code Snippet 25. Calling an Unknown Function with a Casted Pointer

```
1 func:
2   mv a2, a0
3   mv a0, a1
4   mv a1, a2
5   tail unknown
```

Code Snippet 26. Result of Compiling Code Snippet 25 with -O3

```
1 func:
2   addi sp, sp, -64
3   esd ra, 48(sp)
4   ese e1, 56(sp)
5   esd s0, 32(sp)
6   ese e8, 40(sp)
7   addi s0, sp, 64
8   moveee e8, e2
9   esd a0, -40(s0)
10  esd a1, -64(s0)
11  ese e11, -56(s0)
12  eld a0, -64(s0)
13  ele e10, -56(s0)
14  eld a1, -40(s0)
15  call unknown
16  eld s0, 32(sp)
17  ele e8, 40(sp)
18  eld ra, 48(sp)
19  ele e1, 56(sp)
20  addi sp, sp, 64
21  ret
```

Code Snippet 27. Result of Compiling Code Snippet 25 with -O0

limit the effect of inlining and other similar optimizations, but will give us the best estimate for the size overhead of the Zeno compiler.

Overall, there is an average 29.6% size overhead from the Zeno compiler. Figure 12 shows the overhead for individual static libraries inside of newlib. For each static library, the total of the two bars represents the size overhead over the base case LLVM. We break the overhead down and categorize them separately. The first category is the overhead produced by additional Zeno instructions. In this case, we classify `ELE`, `ESE`, `EADDI`, `EADDIX`, and `EADDIE` as the Zeno instructions. The other loads and stores are also Zeno instructions, but they are swapped out one for one with the base RISC-V instructions so we do not consider this to be overhead. The second category is the overhead that is not directly due to a Zeno instruction, these are normal RISC-V instructions.

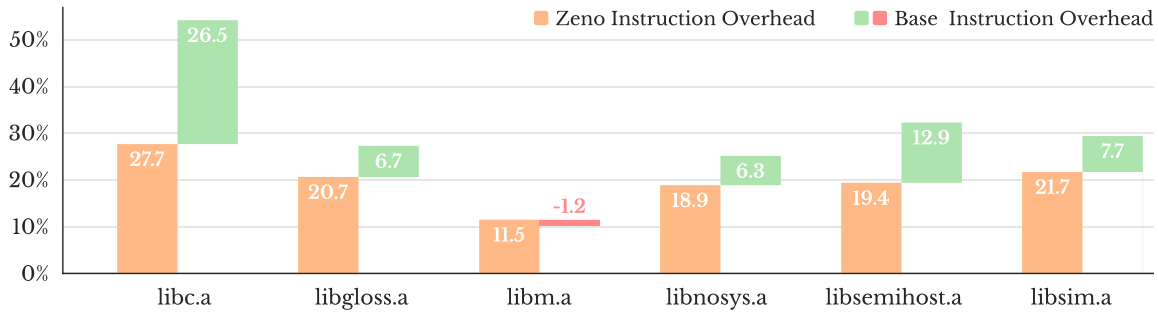


Figure 12. Overheads Produced by Zeno Compiler

The second category, instruction overhead of RISC-V instructions, is the overhead of the missed optimization opportunities caused by adding the Zeno instructions. We hypothesize that because of the extra operations that are needed to preserve pointers, some optimizations opportunities are missed which causes the additional overhead. The overhead caused by just the additional Zeno instructions is an average 19.8% extra lines of code and the overhead caused by missed optimization opportunities is 9.8%.

We note that the best way to decrease the total overhead is to decrease the amount of missed optimization opportunities, but to do this we must decrease the additional Zeno instructions. The two categories are directly tied together, to optimized one is to optimize the other. We also note that the first category, the Zeno instruction overhead, can actually be further split into necessary and unnecessary Zeno instructions. As demonstrated in the static test cases, there are many useless Zeno instructions that get inserted. They do not impact the correctness of the code and have no effect on output of the program, therefore we can remove them. However we do not have a way to further subdivide the category, as if we could determine in the general case which Zeno instructions are necessary and which are not then we would just remove the instruction.

The key takeaway from this is that the required Zeno instruction overhead does

cause a slight degradation in other optimizations, causing additional overhead. The best way to get better code is to allow the compiler to run more aggressive optimizations, which currently Zeno instructions are blocking. Two solutions exist, improve the optimizations themselves to handle extra Zeno instructions and remove useless Zeno instructions.

Note that in all cases but one, adding Zeno instructions incurs an overhead of missed optimization opportunities. But in the case of `libm.a`, it seems to provide more optimization opportunities. This does not make sense in the context of normal, working software. The math library does not primarily operate on pointers, it does computation on values. But the very nature of DAG operators is that the heuristics to apply them are very broad and frequently get applied even if there is no pointer. This normally should not hinder the functionality of the code, as its just ends up being extra pointless register copies. The DAG operators get applied as a result of optimization that inserted potentially illegal operations. Some illegal operations get handled by DAG mapping, while others get handled by the existing compiler. The method of legalization is to replace their result values with an `undef`. An `undef` is a compiler construct, all it means is that the value is unknown. This does not directly translate to a single real operation, rather to the lack of any operation. `undef` frequently gets removed under many optimizations, causing important pieces of code to be 'optimized' away. We hypothesize this is what is occurring in `libm.a`. In short, a compiler bug.

This sort of compiler bug cropped up several times during the development of the compiler. One particularly nasty example was introducing 128 bit values as valid RISC-V values. Because of the poor assumption that a pointer was the largest integer size on the target, the compiler was lowering a struct containing two 64 bit integers

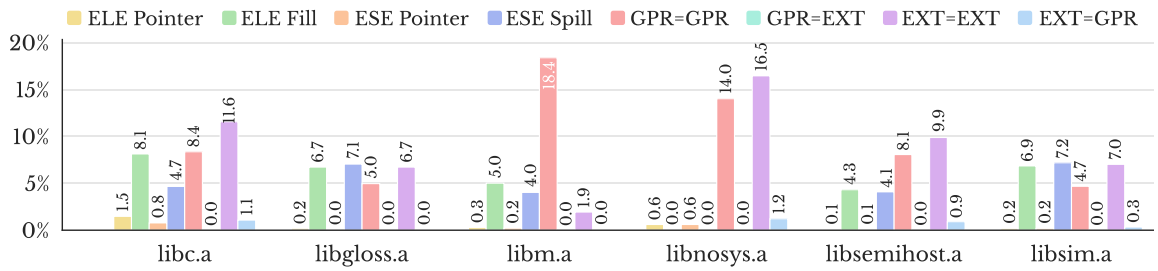


Figure 13. Distribution of Zeno Instructions

to a single 128 bit 'integer'. This 'integer' was then used as a struct indexed by bit shifts (this is a common optimization with valid bit widths, for example converting a struct containing two 32 bit integers to a single 64 bit integer). However, the shift instruction for RISC-V is not defined at 64 bits, so the compiler could 'safely' replace the result of the instruction with `undef`. This particular bug has been patched, but based on the data we hypothesize it is likely that more of these such bugs exist.

We also examine the distribution of Zeno instructions for each static library in Figure 12. Note that the five previously described Zeno instruction categories have now been made into ten. By distinguishing between whether an operation is a spill/fill or a pointer load/store to an arbitrary memory location for `ELE` and `ESE` allows us to reason about the distribution of where these instructions are being used the most. To the list of Zeno instructions is added `MV`, a GPR to GPR copy, since these are often paired with EXT to EXT (`EADDIX`) copies.

In our survey of newlib, the compiler did not generate any EXT to GPR (`EADDI`) copies. This makes intuitive sense, as once a NSID is moved to the extended register file it need not be moved back to the general purpose registers. It will either be copied to another extended register or stored in a memory location directly, we do not perform computation on NSIDs.

Separating out the spills and fills actually shows how much effect they have on total

overhead. The loads and stores of extended registers to arbitrary memory locations is so small as to be almost zero. This demonstrates that there are few double or triple pointers in the sample, we confirm this by investigating the original source code.

For `libc.a`, `libgloss.a`, `libsemihost.a`, and `libsim.a` the distribution of spills, fills, GPR to GPR copies, and EXT to EXT copies is well balanced. We would expect more register copies than spills and fills, since register copies are cheaper, and we do see this pattern. For `libnosys.a` the overhead is dominated by register to register copies, notably not between register files but just shuffling values within the register file. This is expected in small functions that are optimized, as there is little need to save values to the stack when they can be copied to saved registers according to the ABI. `libm.a` is again an outlier, with most of the code being dominated by GPR to GPR copies. This is easier to explain and is likely not the result of a compiler bug, as the math library primarily does not work with pointers and is mostly doing computation heavy tasks.

We also note that on an Out-of-Order core, register renaming would enable many of the register copies to be NOPs. So although we see a large size overhead caused by the register copies, it is unlikely we would see a large runtime effect. This is purely hypothetical and is not verified.

5.5.3 Optimization Opportunities

These graphs, combined with the static tests actually guide us to several ways the Zeno compiler can be improved in the future. The first is to reduce the number of Zeno instructions needed. The static tests demonstrate that the Zeno compiler produces a handful of useless register copies as a result of DAG mapping, we can add

additional optimization passes to solve this. But a key insight from the graphs is that if we reduce the number of added Zeno instructions, we are no longer as limiting of optimization opportunities. This gives us an improved binary almost for free, as by optimizing one thing we are really optimizing two things at once.

There is also a more radical optimization opportunity. Since extended registers follow the same ABI as the general purpose registers, there are a handful of extended registers that if a callee function modifies, it must save and restore the previous state in this register. And since register copies are cheaper and take less instruction space, we could use some of the extended registers to replace our spills and fills. This would be most effective in a leaf function that requires more registers and is doing many spills/fills to the stack to do its computation. From a pure performance and optimization point of view this would be a huge win, particularly on an Out-of-Order core with register renaming. However, there are certain security implications. The extended registers are explicitly there to perform memory addressing, the compiler would essentially be abusing them to gain performance. This does not immediately break any security guarantees of Zeno, but from a security point of view it might be advantageous to not allow this. Currently the Zeno architecture supports copying to and from the extended register file. The Zeno compiler provides this transparently to the user. But a malicious user is perfectly free to write assembly code (or use `__builtin_riscv_zeno_get_nsid`) to arbitrarily read any NSID from the extended register file. We note that the current Zeno architecture does not perform this optimization at all, this is apparent by the lack of compiler generated EXT to GPR (`EADDI`) register copies. In the future, the Zeno architecture may remove the ability to copy from the extended registers back to the general purpose registers from software (or make it a privileged operation). This would enforce better security as NSIDs are

no longer visible to an attacker seeking to learn information about the system. So we state this optimization is possible, but likely has undesirable security impacts.

Chapter 6

CONCLUSION

6.1 Summary of Work

In this thesis we described a small subset capability architectures, and generalized them into two distinct types. Capability architectures with a single, unbreakable capability have been well researched and have mature compiler toolchains developed for them. Capability architectures with a separable capability that can only pass through the data path as separate pieces are a still relatively unexplored area. Emerging capability architectures are using this implementing, but require new software tools and compiler techniques to develop for them and run software.

We then introduced a common language for discussing compilers to more succinctly describe the following compilation techniques. To describe our compilation techniques, a hypothetical generic capability architecture was detailed. We presented two approaches, DAG mapping and pseudo instructions, which complement each other to compile code with separable capabilities transparently. A further discussion was raised regarding the semantics of a capability architecture. Lastly, we described several ways to improve the usability and developer quality of life for our compilation techniques.

To demonstrate the validity of the techniques, we implemented a subset of them in a compiler for the Zeno architecture. This compiler utilized extensive DAG mapping and pseudo instructions to build a Zeno compiler on top of the RISC-V backend in LLVM. We evaluated the compiler statically through rigours test cases and an analysis of the distribution of instructions in a given binary.

6.2 Key Takeaways

The primary takeaway is that DAG mapping is viable technique to produce correct code for separable capability architectures. Although certain inefficiencies do exist, it produces correct code that preserves capabilities. We also showed that many of the inefficiencies introduced can be easily optimized away. This is an important part of this work, when is it okay to optimize a given section of code and when is it not.

Another important takeaway is not directly related to DAG mapping, but rather to software engineering as a whole. Certain assumptions can be dangerous or introduce unforeseen issues later on. Bad assumptions cause software developers to not check the bounds of their array, leading to buffer overflow attacks that make capability architectures a requirement. Furthermore, much of this work payed the price of having to fix poor compiler assumptions. Many pieces of software and the compiler itself assume that a pointer is 64 bits, but as capability architectures become more prevalent this is not going to be true much longer. The software community did not learn their lesson on the 32 bit pointer to 64 bit pointer transition, hopefully on the next change the poor assumptions are solved.

6.3 Future Work

There is quite a bit of future engineering work for the Zeno compiler. As discussed, there are many obvious opportunities for optimization that will vastly improve the Zeno compiler. The examples presented in this work are just the tip of the iceberg. In future work, it may also be interesting to investigate what other assumptions are made by compilers and programming languages that may be broken in the coming years.

Hardware development is making a huge resurgence; as new and different hardware is developed it may break assumptions that have hidden for years.

Research on capability architectures is ongoing, but much of the work is focused on the hardware. We believe future research should consider the software and tooling impacts capabilities will have.

REFERENCES

[1] *2021 CWE Top 25 Most Dangerous Software Weaknesses*. 2021. URL: https://cwe.mitre.org/top25/archive/2021/2021%5C_cwe%5C_top25.html.

[2] Jacob Abraham, Alan Ehret, and Michel Kinsy. "A Compiler for Transparent Namespace-Based Access Control for the Zeno Architecture". In: *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. 2022, pp. 169–178. DOI: 10.1109/SEED55351.2022.00022.

[3] James P Anderson. *Computer Security Technology Planning Study Volume I and II ESD-TR-73-51*. Tech. rep. Hanscom Field, Bedford, MA: Electronic Systems Division, Air Force Systems Command, Oct. 1972.

[4] Claudio Canella et al. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *CoRR* abs/1811.05441 (2018). arXiv: 1811.05441. URL: http://arxiv.org/abs/1811.05441.

[5] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. "Hardware Support for Fast Capability-Based Addressing". In: *SIGPLAN Not.* 29.11 (Nov. 1994), pp. 319–327. DOI: 10.1145/195470.195579. URL: https://doi.org/10.1145/195470.195579.

[6] Ctsrd-Cheri. *CTSRD-Cheri/LLVM-project: Fork of LLVM adding Cheri Support*. URL: https://github.com/CTSRD-CHERI/llvm-project.

[7] *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer*. July 2006. URL: https://cwe.mitre.org/data/definitions/119.html.

[8] *CWE-125: Out-of-bounds Read*. July 2006. URL: https://cwe.mitre.org/data/definitions/125.html.

[9] *CWE-416: Use After Free*. July 2006. URL: https://cwe.mitre.org/data/definitions/416.html.

[10] *CWE-787: Out-of-bounds Write*. Oct. 2009. URL: https://cwe.mitre.org/data/definitions/787.html.

[11] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. DOI: 10.1145/115372.115320. URL: https://doi.org/10.1145/115372.115320.

[12] Davidchisnall et al. *Is it time to start upstreaming the Cheri support to LLVM?* Feb. 2022. URL: https://discourse.llvm.org/t/is-it-time-to-start-upstreaming-the-cheri-support-to-llvm/60032.

[13] Brooks Davis et al. "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 379–393. DOI: 10.1145/3297858.3304042.

[14] Joe Devietti et al. "Hardbound: Architectural Support for Spatial Safety of the C Programming Language". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 103–114. DOI: 10.1145/1346281.1346295.

[15] Alan Ehret et al. *Zeno: A Scalable Capability-Based Secure Architecture*. 2022. DOI: 10.48550/ARXIV.2208.09800.

[16] Richard Grisenthwaite. *Morello research program hits major milestone with hardware now available for testing*. Jan. 2022. URL: https://www.arm.com/company/news/2022/01/morello-research-program-hits-major-milestone-with-hardware-now-available-for-testing.

[17] *Intel 64 and IA-32 Architectures Software Developer Manual*. Intel. 2022.

[18] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. Fourth. June 2018, p. 520. URL: https://www.iso.org/standard/74528.html.

[19] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: San Jose, CA, USA, Mar. 2004, pp. 75–88.

[20] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[21] John Leidel et al. *RISC-V Extended Addressing Architecture Extension Specification Codenamed: xBGAS*. Version 0.0.6. Oct. 2019. URL: https://github.com/tactcomplabs/xbgas-archspec.

[22] Kyung-Suk Lhee and Steve J. Chapin. "Buffer overflow and format string overflow vulnerabilities". In: *Software: Practice and Experience* 33.5 (2003), pp. 423–460. DOI: 10.1002/spe.515.

[23] *Linux Kernel Source*. URL: https://github.com/torvalds/linux.

[24] *LLVM Documentation*. LLVM Project. 2022. URL: https://llvm.org/docs/index.html.

[25] Carlos O'Donell and Martin Sebor. *Updated Field Experience With Annex K - Bounds Checking Interfaces*. Sept. 2015. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1969.htm.

[26] Oleksii Oleksenko et al. "Intel MPX Explained". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2.2 (June 2018), pp. 1–30. DOI: 10.1145/3224423. URL: https://doi.org/10.1145/3224423.

[27] Aleph One. "Smashing the Stack for Fun and Profit". In: *Phrack* 7.49 (Nov. 1996). URL: http://phrack.org/issues/49/14.html.

[28] C. W. Otterstad. "A brief evaluation of Intel®MPX". In: *2015 Annual IEEE Systems Conference (SysCon) Proceedings*. 2015, pp. 1–7. DOI: 10.1109/SYSCON.2015.7116720.

[29] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global Value Numbers and Redundant Computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 12–27. DOI: 10.1145/73560.73562. URL: https://doi.org/10.1145/73560.73562.

[30] Kostya Serebryany et al. *Memory Tagging and how it improves C/C++ memory safety*. 2018. DOI: 10.48550/ARXIV.1802.09517. URL: https://arxiv.org/abs/1802.09517.

[31] R. Sharifi and A. Venkat. "CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 762–775. DOI: 10.1109/ISCA45697.2020.00068.

[32] Xi Wang et al. "xBGAS: A Global Address Space Extension on RISC-V for High Performance Computing". In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021, pp. 454–463. DOI: 10.1109/IPDPS49936.2021.00054.

[33] Andrew Waterman and Krste ASANOVI C. "The RISC-V Instruction Set Manual, volume II: Privileged Architecture". In: *CS Division, EECE Department, University of California, Berkeley (May 2017)* (2017).

[34] Robert Watson, Ken Hamer-Hodges, and Geoffrey Stagg. *Arm releases experimental CHERI-enabled Morello board as part of £187M UKRI Digital Security by Design programme*. Jan. 2022. URL: https://www.lightbluetouchpaper.org/2022/01/20/arm-releases-experimental-cheri-enabled-morello-board-as-part-of-187m-ukri-digital-security-by-design-programme/.

[35] Robert N. M. Watson et al. *Capability hardware enhanced RISC instructions: CHERI instruction-set architecture*. Tech. rep. University of Cambridge, Computer Laboratory, 2015.

[36] Robert N. M. Watson et al. *CHERI C/C++ Programming Guide*. Tech. rep. UCAM-CL-TR-947. University of Cambridge, Computer Laboratory, June 2020. DOI: 10.48456/tr-947. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf.

[37] Robert N.M. Watson et al. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 20–37. DOI: 10.1109/SP.2015.9.

[38] Andrew Watterman and Krste Asanovic. *The RISC-V Instruction Set Manual-Volume I: User-Level ISA*. Tech. rep. Version 2.2. RISC-V Foundation, May 2017.

[39] Jonathan Woodruff et al. "Cheri concentrate: Practical compressed capabilities". In: *IEEE Transactions on Computers* 68.10 (2019), pp. 1455–1469.

[40] Jonathan Woodruff et al. "The CHERI capability model: Revisiting RISC in an age of risk". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE. 2014, pp. 457–468.

[41] Hongyan Xia et al. "CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 545–557.