TÉCNICO LISBOA

Supporting Raw Data Copies in the LLVM Intermediate Representation

Pedro Cerqueira Lobo

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Nuno Claudino Pereira Lopes

Examination Committee

Chairperson: Prof. Paolo Romano
Supervisor: Prof. Nuno Claudino Pereira Lopes
Member of the Committee: Prof. José Faustino Fragoso Femenin dos Santos

November 2025

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First and foremost, I wish to express my gratitude to my dissertation supervisor, Prof. Nuno Lopes, for his invaluable expertise, mentorship, and dedication. His insightful guidance and feedback were crucial in shaping both the direction and quality of our work. Beyond providing critical support during some of the most challenging phases of this thesis, his encouragement and engagement greatly contributed to making the entire process both rewarding and genuinely enjoyable.

In addition, I am deeply grateful to my family for their unconditional support and encouragement throughout this journey. A special acknowledgment to my godmother, who always looks after me and encourages me to take breaks from work.

My sincere thanks also go to my friends and colleagues, to whom I have explained countless times what my thesis was about, for their patience, support, and the enjoyable dinners and moments we shared throughout this journey.

Last but not least, I would like to extend my sincere acknowledgments to Eng. George Mitenkov, who initiated work on this topic as part of a Google Summer of Code project, back in 2021. His accomplishments during that period laid an important groundwork for this thesis. Moreover, the excellence and clarity reflected in his writing on the topic addressed by this work not only greatly enhanced my understanding of the problem, but also inspired me to pursue a similar high standard in my own work.

To everyone who contributed to this work, or to my education and personal growth, whether directly mentioned or not, I offer my sincere gratitude. Your support, guidance, and encouragement have played an essential role in shaping me into the person I am today. Thank you.

Abstract

LLVM is a widely used compiler infrastructure, leveraged as a code generation backend by several programming languages. One of LLVM's longstanding problems is the absence of a type capable of representing raw memory in its Intermediate Representation (IR), akin to `unsigned char` in C or `std::byte` in C++. This limitation is both the root cause of several unsound optimizations currently performed by LLVM, and an obstacle to the native implementation of memory-related primitives in the IR, including `memcpy`, `memmove`, and `memcmp`.

Currently, as a workaround, frontends load raw memory values as plain integers, inadvertently performing type punning and discarding pointer provenance information. Allowing such type punnings would break LLVM's alias analysis algorithms due to potential aliases arising from every integer load. Moreover, such type punnings are unsound, as loading memory data containing unspecified values, such as padding bits, taints the result. This renders copies of such data impossible to perform without resorting to opaque intrinsics.

This thesis proposes the addition of a byte type to LLVM IR. The new type can represent raw memory values, avoiding the aforementioned compiler-introduced type punning and enabling the implementation of the previously mentioned memory-related primitives. We use the byte type to fix existing optimizations and introduce new ones. We also add support for the byte type in Alive2, verifying both the reworked and newly implemented optimizations. Finally, we show that the proposed solution fixes known miscompilations, while having a minimal impact on both run-time and compile-time performance.

Keywords

LLVM; Raw Memory Values; Memory Model; Pointer Provenance.

Resumo

O LLVM é uma infraestrutura de compiladores amplamente conhecida, utilizada na geração de código de diversas linguagens de programação, como o C, C++, Rust e Swift. Um dos seus grandes problemas é a ausência de um tipo capaz de representar um valor genérico em memória na representação intermédia, de forma semelhante ao `unsigned char` em C ou ao `std::byte` em C++. A ausência deste tipo é a causa fundamental de muitas otimizações incorretas e um obstáculo à implementação de certas funções na representação intermédia, incluindo o `memcpy`, `memmove` e `memcmp`.

Atualmente, como solução alternativa, os *frontends* carregam valores da memória como inteiros, convertendo implicitamente inteiros em ponteiros, o que descarta a proveniência dos ponteiros. Permitir estas conversões implícitas comprometeria os algoritmos de análise de ponteiros do LLVM. Além disso, tais conversões são incorretas, pois ao carregar dados de memória que contêm valores não especificados, como bits de preenchimento, o resultado é invalidado, tornando impossíveis as cópias desses dados sem recorrer a funções especiais, definidas de forma opaca.

Esta dissertação propõe a adição de um *byte type* à representação intermédia do LLVM, capaz de representar valores genéricos em memória, evitando conversões implícitas e permitindo a implementação das funções anteriormente referidas. Utilizamos o *byte type* para corrigir e introduzir novas otimizações. Adicionamos o novo tipo ao Alive2, automatizando a verificação das otimizações corrigidas e implementadas. Finalmente, mostramos que a introdução do novo tipo no LLVM tem um impacto desprezível no tempo de compilação e de execução.

Palavras Chave

LLVM; Valores de Memória Genéricos; Modelo de Memória; Ponteiros.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

DAG Directed Acyclic Graph

IR Intermediate Representation

PVI Provenance-via-Integers

PNVI Provenance-not-via-Integers

UB Undefined Behavior

SSA Static Single-Assignment

SIMD Single Instruction, Multiple Data

SLP Superword-Level Parallelism

1

Introduction

Contents

Compilers occupy an important role in the software development process, translating high-level languages into machine code that can be executed by a computer. An optimizing compiler, in particular, aims to improve a given performance metric of the generated code through various analysis and optimization techniques. These code transformations are typically performed on an Intermediate Representation (IR), abstracting away the details of the source language and target architecture.

Employing an IR simplifies the design of an optimizing compiler, promoting modularity. Frontends lower the source language into the IR, while backends produce machine code from the IR, targeting a particular architecture. This enables the reuse of the optimizer by different frontends and backends. Moreover, an IR provides compiler developers the flexibility to design and extend it in ways that facilitate optimization. In particular, new IR constructs can be introduced as needed.

The IR of an optimizing compiler should be able to represent high-level constructs found in the source language, while simultaneously remaining close to the target machine, modeling low-level operations. However, it should remain source and target-agnostic, avoiding assumptions about particular source

languages or target architectures. Moreover, the semantics of the IR should be clearly specified, allowing compiler developers to understand and agree on the meaning given to its constructs. Nonetheless, misunderstandings or disagreements have been known to cause miscompilations, where the compiler generates incorrect code, not preserving the intended semantics of the source program.

In 2019, a bug in GCC affecting the Power9 architecture incorrectly optimized away multiple calls to a random number generator function used in a cryptographic context, assuming it to be a pure function. This resulted in the same value being returned across distinct invocations.[1] In 2016, a bug in LLVM caused Clang to miscompile itself, as two optimization passes adopted conflicting semantics when branching on an undefined value.[2]

Over the past decade, numerous research efforts have focused on formalizing the LLVM IR [1–8]. In parallel, tools have been developed to automate the verification of compiler analyses and optimizations [9–11], and to programmatically generate and reduce test cases that trigger miscompilations [12, 13]. Collectively, these initiatives have played a significant role in improving the design of compiler IRs and reducing miscompilation bugs in modern compilers.

One of the longstanding limitations of the LLVM IR, pinpointed by Alive2 [14], is its inability to represent and manipulate raw memory values. Currently, memory loads of raw bytes are performed through appropriately sized integer types. However, these are incapable of representing an arbitrary memory value, namely pointers and padded values. The absence of a dedicated type to represent and manipulate raw memory hinders the native implementation of memory-related intrinsics such as `memcpy`, `memmove` and `memcmp` in the IR. Moreover, it introduces additional friction when loading and converting memory values to other types, leading to implicit conversions that are hard to identify and reason about. The core problems stemming from the absence of a proper type to access and manipulate raw memory, directly addressed by our proposal and explored throughout the remainder of this section, are summarized as follows:

1. Integers do not track provenance information, which corresponds to metadata attached to pointers, used to justify optimizations, rendering them unsuitable to represent a pointer value.

2. Loads through integer types spread `poison` values, used to represent padding bits, tainting the load result if the memory value contains at least one `poison` bit.

According to the LLVM Language Reference Manual,[3] which acts as the specification for its IR, "*LLVM IR does not associate types with memory. The result type of a load merely indicates the size and alignment of the memory from which to load. (...) The first operand type of a store similarly only indicates the size and alignment of the store*". On one hand, this statement suggests that a load/store of a value

[1] https://nvd.nist.gov/vuln/detail/cve-2019-15847
[2] https://github.com/llvm/llvm-project/issues/31000, https://github.com/llvm/llvm-project/issues/27880
[3] https://llvm.org/docs/LangRef.html

of any given type is equivalent to a load/store of the same value, coerced to an integer. On the other hand, this notion is incompatible with the provenance-aware memory model [5] used by LLVM, which associates provenance with pointers. Such provenance information is stored as metadata alongside the pointer, allowing it to be loaded and stored in memory. While pointers capture this metadata, integers do not, preventing the latter from accurately modeling a pointer value.

The two aforementioned memory models are in conflict, as loading a pointer from memory through an integer type introduces implicit type punning, inadvertently casting the pointer to an integer and discarding the pointer's provenance. This disagreement is the root cause of several end-to-end miscompilations,[4] as several optimizations do not correctly account for the compiler-introduced type punning.

Currently, Alive2 uses `poison`, one of LLVM's Undefined Behavior (UB) values, to represent the result of loading a pointer from memory through an integer type. Therefore, loads of pointers through any type other than a pointer type fail to accurately recreate the original memory value. Listing 1 shows an unsound transformation,[5] where a pointer is stored and loaded back from memory. On the left, the load is performed through a pointer type, yielding the same, previously stored value. On the right, the load through an integer type implicitly casts the pointer to an integer and returns `poison`.

```
store ptr %v, ptr %ptr
%p = load ptr, ptr %ptr
```

```
store ptr %v, ptr %ptr
%i = load i64, ptr %ptr      ; poison
%p = inttoptr i64 %i to ptr
```

Listing 1: Invalid transformation, as loading a pointer through an integer type returns `poison`

This problem is further exacerbated by the fact that loading pointers through integer types can cause pointer escapes to go unnoticed by alias analysis. By discarding provenance information, pointer addresses, represented as plain integers, may be leaked to other functions and used to craft an aliasing pointer, via an integer-to-pointer cast, while bypassing escape analysis. Once alias analysis is compromised, simple optimizations that rely on the absence of aliasing are no longer justified, compromising the correctness of the entire compilation process.

A second problem with the current semantics of the IR lies in loading padded values. LLVM represents padding bits with the `poison` value. If at least one of the bits stored in memory is `poison`, loads of this value through an integer type taint the load result, returning `poison`. This occurs as integer types track `poison` on a per-value basis, failing to provide enough granularity to represent individual `poison` bits. In Listing 2, a structure with two fields is defined. In order to satisfy alignment requirements, the compiler inserts three bytes of padding between the two fields. The `assign` function, on the left, is lowered as a call to `memcpy`, which is then optimized into an integer load/store pair, as shown on the right. Loading the padded value through the `i64` integer type taints the load result, and the `poison` value is then stored to memory.

[4] https://github.com/llvm/llvm-project/issues/33896
[5] https://alive2.llvm.org/ce/z/VUTsdW

```c
typedef struct {
  char a;
  int b;
} S;

void assign(S *a, S *b) {
  *a = *b;
}
```

```llvm
%struct.S = type { i8, i32 }

define void @assign(ptr %a, ptr %b) {
  %s = load i64, ptr %b
  store i64 %s, ptr %a
  ret void
}
```

Listing 2: Lowering of a C program copying a padded structure (left) to LLVM IR (right)

Moreover, this inflexibility presented by integer types leads to subtle issues that are often overlooked in the implementation of compiler passes. The LLVM Language Reference[6] defines the `bitcast` instruction as a "*no-op cast because no bits change with this conversion*". Nonetheless, while scalar types track `poison` on a per-value basis, vector types do it on a per-lane basis. This introduces potential pitfalls when casting vector types to non-vector types, as the cast can inadvertently taint non-`poison` lanes. In Listing 3, if the first lane of `%v` is `poison`, the result of casting the vector to an `i64` value is also `poison`, regardless of the value of the second lane.

```llvm
%v = load <2 x i32>, ptr %ptr       ; <i32 poison, i32 42>
%c = bitcast <2 x i32> %v to i64    ; i64 poison
```

Listing 3: A `bitcast` from an integer vector to scalar type spreads `poison` values

Although covered by the LLVM Language Reference ("*the [`bitcast`] conversion is done as if the value had been stored to memory and read back as [the destination type]*"), this duality in the value representation between vector and scalar integer types constitutes a corner case that is not widely contemplated and often unnecessarily introduces UB.

Low-level languages, such as C and C++, solve the aforementioned problems by providing proper types to access raw memory data. The `unsigned char` type can be used to access and manipulate an object binary representation, both in C[7] and C++.[8] This enables the implementation of memory-related primitives like `memcpy`, as values are copied byte-by-byte, without implicit casts. Moreover, both languages specify that, under strict aliasing rules, any pointer may alias with a pointer to a character type,[9] allowing pointers to be copied without circumventing alias analysis.

In an effort to improve type safety, C++17 introduced the `std::byte` type. While the built-in character types perform raw memory accesses, these are also arithmetic types, enabling users to mistakenly perform arithmetic on values that should be treated as bytes. Thus, the `std::byte` type inherits the same raw memory access capabilities as the built-in character types, but disallows arithmetic operations.

[6] https://llvm.org/docs/LangRef.html#bitcast-to-instruction

[7] *Values stored in non-bit-field objects of any other object type consist of n x `CHAR_BIT` bits, where `n` is the size of an object of that type, in bytes. The value may be copied into an object of type `unsigned char [n]` (e.g., by `memcpy`); the resulting set of bytes is called the object representation of the value.* (C99 Standard, 6.2.6.1.4)

[8] *The underlying bytes making up the object can be copied into an array of `char`, `unsigned char`, or `std::byte`. If the content of that array is copied back into the object, the object shall subsequently hold its original value.* (C++23 Standard, 6.8.1.2)

[9] *An object shall have its stored value accessed only by an lvalue expression that has one of the following types (...) a character type.* (C99 Standard, 6.5.7)

1.1 Contributions

In this work, we add a byte type to LLVM IR. Memory loads through the byte type do not introduce implicit casts, yielding the value's raw representation. This allows the byte type to represent both pointer and non-pointer values. Moreover, the new type provides the necessary granularity to represent individual `poison` bits, such that loads of padded values through the byte type do not taint the load result.

We use the byte type to fix existing optimizations and implement new ones. Its addition to the IR also enables the native implementation of memory-related intrinsics, including `memcpy`, `memmove`, and `memcmp`. We lower Clang's character types, which include `unsigned char`, `char`, `signed char`, and the `std::byte` type to the byte type, preserving their raw memory access semantics and fixing existing miscompilations in LLVM. Moreover, we add support for the new type to Alive2, verifying the correctness of both the reworked and newly implemented optimizations.

Finally, we show that the introduction of the byte type to LLVM has a negligible impact on the size and performance of the generate code, compilation time and peak memory usage of the compiler. We also validate the correctness of the implemented solution by validating the compilation of two benchmark programs using translation validation. Furthermore, we show that the implementation of the new type in Alive2 does not adversely affect validation time.

1.2 Outline

This document is structured as follows. In Chapter 2, we provide a high-level overview of the LLVM project and introduce the reader to LLVM IR, showcasing lowerings of simple C programs to the IR. Then, in Chapter 3, we introduce the concept of UB, discuss its role in optimizing compilers and explore how it is incorporated within LLVM. Chapter 4 explores the two main types of memory models aiming to reconcile high-level optimizations with low-level idioms, with respect to their representation of pointers.

In Chapter 5, we go over some of the related work, analyzing the most prominent memory models used by optimizing compilers and addressing a proposal to extend LLVM with support for C's `restrict` keyword. Chapter 6 describes our proposal in greater detail, while Chapter 7 explores some of the implementation details, both in LLVM/Clang and Alive2. Chapter 8 evaluates our solution and provides a detailed analysis on the obtained results. Finally, Chapter 9 concludes with a summary of achievements and future work recommendations.

2

LLVM

Contents

LLVM is a widely adopted compiler infrastructure, providing a robust foundation for building a broad spectrum of compiler and language tools, including debuggers, static analyzers, sanitizers and linkers. Its modular architecture makes it a compelling code generation target for a wide range of programming language implementations. In particular, LLVM is used as a code generation backend by several well-known programming languages, including C, C++, Rust and Swift.

In this chapter, we provide a high-level overview of the LLVM project. In Section 2.1, we introduce LLVM's three-phase architecture, outlining the major phases of its compilation pipeline. Then, in Section 2.2, we introduce Static Single-Assignment (SSA)-based representations and explore LLVM's IR, examining its type system, a subset of its most relevant instructions, and its extension mechanisms, which allow source languages to encode high-level information without directly implementing new IR constructs.

2.1 LLVM Project

LLVM was initially created with the goal of supporting static and dynamic compilation strategies for multiple programming languages. Since its inception, LLVM has grown to accommodate various sub-projects. These include its core libraries, providing a target-independent optimizer and code generation utilities targeting several CPU architectures, the Clang C/C++ compiler frontend and the LLDB debugger.

LLVM follows a three-phase architecture, consisting of a frontend, middle-end, and backend. The frontend is responsible for lexing and parsing source code, often performing type checking and semantic analysis, ultimately converting the high-level programming language into LLVM IR. The LLVM project includes Clang, a frontend for C and C++. Other projects take advantage of LLVM's code generation capabilities by providing a custom frontend, such as Rust with `rustc` and Swift with `swiftc`.

Once LLVM IR is emitted, it is processed by the middle-end or optimizer, which applies a series of source and target-independent transformations, supported by various analysis passes. The main goals of the optimizer are to simplify, canonicalize and optimize the IR.

Finally, the backend translates the optimized LLVM IR into machine code for a specific target architecture, performing instruction selection, register allocation and instruction scheduling. The backend also performs target-specific optimizations as, typically, the middle-end is target-independent.

By decoupling the frontend, middle-end and backend components, LLVM enables the reuse of its analysis and optimization infrastructure. This allows the compiler to be easily retargeted, that is, to implement multiple backends, allowing a single source language to target multiple architectures. Similarly, a common code representation also enables frontends to reuse existing backends. This architecture, illustrated in Figure 2.1, facilitates the rapid development of new programming languages, as much of the heavy lifting in optimization and code generation can be shared across different implementations of various programming languages.

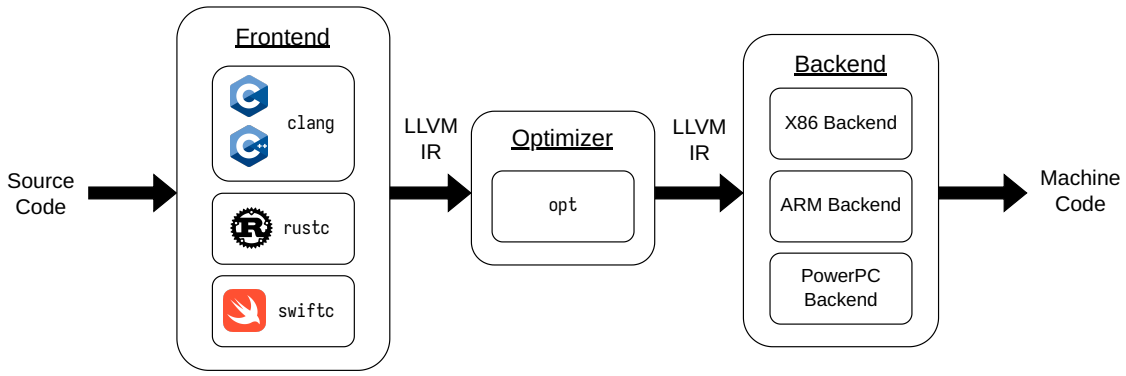


Figure 2.1: LLVM's three-phase architecture. The frontend parses source code and emits LLVM IR. The middle-end applies target-independent optimizations. The backend generates machine code for a specific target architecture.

2.2 LLVM IR

LLVM's IR is the common code representation used throughout all target-independent phases of the compilation process. It aims to model low-level operations, while cleanly representing the high-level idioms present in most source languages without directly implementing language-specific features. It is statically typed and supports structures, arrays and vectors as first-class types.

Throughout the remainder of this section, we show the lowering of simple C programs to LLVM IR, introduce the concept of SSA-based representation, a fundamental property of the IR, and explore its most relevant types, instructions and extension mechanisms.

2.2.1 Program Anatomy

Programs in LLVM IR are composed of global variables and functions. Functions correspond to a set of basic blocks, which are linear sequences of instructions ending with a terminator instruction, such as a return or branch instruction. Terminator instructions define the control flow of the program by transferring execution to another basic block.

Listing 4 shows the lowering of a simple C function to LLVM IR, containing local variables, preceded by %, and global variables, such as functions, preceded by @. This function takes an integer argument %x, and returns a value of the same type, i32, a 32-bit integer. Structurally, this function is composed of a single basic block, delimited by the mul and ret instructions. The latter is a terminator instruction. LLVM IR is strongly typed, requiring explicit type annotations. This is evident in the mul and add instructions, which specify that their operands are of type i32. These instructions may encode additional information in the form of flags, used by analysis and optimization passes. The nsw flag specifies that an erroneous value is returned if signed integer overflow occurs, encoding the semantics of the C programming language.

```c
int double_plus_four(int x) {
  return 2 * x + 4;
}
```

```
define i32 @double_plus_four(i32 %x) {
  %double = mul nsw i32 2, %x
  %res    = add nsw i32 %double, 4
  ret i32 %res
}
```

Listing 4: An example program in C (left) and the corresponding LLVM IR (right)

2.2.2 Single Static Assignment

LLVM IR is an SSA-based representation [15–17], in which each variable is defined exactly once, hence single static assignment. Each definition has a distinct name and dominates all of its uses, that is, all paths from the entry of the function to a specific use must traverse the corresponding definition. For instance, the function in Listing 5 is ill-formed, as the definition of %x, on line 1, does not dominate its

use within the same instruction. On the other hand, the previous example in Listing 4 is well-formed, as the definition of %double dominates its use in the add instruction.

```
1  define i32 @foo(i32 %x) {
2    %x = add i32 %x, 1
3    ret i32 %x
4  }
```

Listing 5: Ill-formed LLVM function that violates SSA form, as the definition of %x does not dominate its use

The single-assignment property of SSA-based representation ensures that values are never dangling or redefined, as a definition must be reachable from any particular use of a given value. This simplicity favors both the implementation and performance of analysis and transformation passes in an optimizing compiler. However, the single-assignment constraint conflicts with in-place mutation, posing a challenge when needing to update the value of the same variable in two distinct program paths, such as within both branches of a conditional instruction. This is addressed by introducing a special SSA construct, the ϕ-node (represented in LLVM by the phi instruction), where control-flow paths meet.

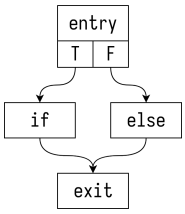
In the C program shown in Figure 2.2, variable a is either doubled or squared, depending on the traversed control flow path. The corresponding LLVM IR program, on the right, computes the result of the addition and multiplication, on lines 6 and 10, respectively. These results are assigned to distinct SSA values, %a.1 and %a.2, each representing a different version of the original variable a. Since its value is used after the conditional, the program must reconcile the two definitions. This is achieved by introducing a phi instruction in the %exit block. This instruction is parameterized by multiple values, each associated with a single predecessor basic block. A phi instruction must have exactly one value for every predecessor of the given basic block. It yields the value associated with the basic block from which control flow is reached. If execution reaches the %exit block from the %if block, %a.3 gets assigned the value of %a.1. Otherwise, control must flow from the %else block and the assigned value is %a.2.

The phi instruction effectively introduces a value assignment at control flow merge points. This enables SSA-based representations to model variable mutation while preserving the single-assignment property. A single basic block can have multiple phi instructions, all of which are executed concurrently, before the remaining instructions. This highlights the functional nature of SSA-based representations, as ϕ-nodes can be interpreted as named arguments to basic blocks, as opposed to special instructions that concurrently execute at the entry of basic blocks.

2.2.3 Types and Instructions

The type system constitutes an important feature of the IR, enabling analysis and optimizations that would otherwise be infeasible or significantly more costly to perform. LLVM's type system offers the expected primitive types found in most source languages, such as integers, pointers, and floating-points. It also offers more complex constructs, like aggregate types, including arrays (such as [4 x i8]), and

```c
int foo(bool cond) {
  int a = 4;
  if (cond) a = a + a;
  else a = a * a;
  return a;
}
```




```llvm
define i32 @foo(i1 %cond) {
entry:
  br i1 %cond, label %if, label %else

if:
  %a.1 = add nsw i32 4, 4
  br label %exit

else:
  %a.2 = mul nsw i32 4, 4
  br label %exit

exit:
  %a.3 = phi i32 [ %a.1, %if ], [ %a.2, %else ]
  ret i32 %a.3
}
```

Figure 2.2: An example program in C (top-left), its CFG (bottom-left), and the corresponding LLVM IR (right)

structure types (such as `{ i32, ptr }`), functions, and vector types (such as `<8 x float>`). The latter enables Single Instruction, Multiple Data (SIMD) instructions.

Pointers are represented by the opaque type `ptr`. These are used in memory-related operations, such as `load` and `store` instructions. While `store` instructions are parameterized by the value to store and a memory location (in the form of a pointer), `load` instructions specify the type of the loaded value and the memory location from which to load. The use of these instructions is illustrated in Listing 6.

```c
int foo(int *ptr, int val) {
  *ptr = val;
  return *ptr;
}
```

```llvm
define i32 @foo(ptr %ptr, i32 %val) {
  store i32 %val, ptr %ptr
  %1 = load i32, ptr %ptr
  ret i32 %1
}
```

Listing 6: A program in C (left) storing and loading the value of a pointer and the corresponding LLVM IR (right)

LLVM IR is strongly typed, requiring explicit type conversions. These are handled by a set of conversion instructions, each parameterized by a single operand and a target type. Standard integer conversion operations are supported, such as sign and zero extension. Integer truncation is also supported. Conversions between equally sized types use the `bitcast` instruction, which reinterprets the bit pattern of a value without modifying it. Conversions of pointers to other types constitute an exception, as these are handled by other instructions. Pointers can be converted to integers through the `ptrtoint` instruction. Conversely, integers can be cast to pointers using the `inttoptr` instruction. Listing 7 shows the lowering of a Rust program to LLVM IR, using some of these conversion instructions.

The `getelementptr` instruction is used to compute the address of a sub-element within an aggregate type. In Listing 8, the type alias on line 1 assigns a structure with three members, a 32-bit integer, an array of four 8-bit integers, and a vector of two `double` values, to the name `%S`. The `getelementptr` instruction, on line 4, computes the address of the third element within the array field of the structure.

```rust
fn foo(s: i32, u: u32, f: f32,
       ptr: *mut i32, l: u64) {
  let a = s as i64;
  let b = u as u64;
  let c = s as i16;
  // Reinterpret `f` as an int
  let d: i32 = unsafe {
    std::mem::transmute(f)
  };
  let e = ptr as i64;
  let p: *mut i32 = l as *mut i32;
}
```

```
define void @foo(
  i32 %s, i32 %u, float %f,
  ptr %ptr, i64 %l
) {
  %a = sext i32 %s to i64
  %b = zext i32 %u to i64
  %c = trunc i32 %s to i16
  %d = bitcast float %f to i32
  %e = ptrtoint ptr %ptr to i64
  %p = inttoptr i64 %l to ptr
  ret void
}
```

Listing 7: A program in Rust (left) performing conversion operations and the corresponding LLVM IR (right)

The first index always indexes the pointer value given as the second argument to the instruction. The following index accesses the array, the second field of the structure. The last index selects the third element within the array. The `inbounds` flag constrains the resulting pointer to be in bounds of the allocated object. If the address computation goes out of bounds, an erroneous value is returned.

```
%S = type { i32, [4 x i8], <2 x double> }

define ptr @foo(ptr %s) {
  %arrayidx = getelementptr inbounds %S, ptr %s, i32 0, i32 1, i32 2
  ret ptr %arrayidx
}
```

Listing 8: The `getelementptr` instruction computes the address of the third element of a structure's array field

2.2.4 IR Extension Mechanisms

LLVM's type system is not expressive enough to represent some constraints presented by high-level languages, such as signed integer overflow behavior, or inferred through program analysis, such as value ranges, aliasing behavior, or alignment guarantees. To that end, special constructs extend the expressive power of the IR by adding semantic information meant to be used by analysis and optimization passes. These constructs are divided among the following three categories.

1. **Attributes:** These encode properties of function parameters and return values that are part of a function definition or callsite. The `nonnull` attribute in Listing 9 is applied to pointer `%ptr`, indicating that this argument is never `null`. If the function is called with a `null` argument, the parameter is assigned an erroneous value.

```
define i32 @foo(ptr nonnull %ptr) {
  %val = load i32, ptr %ptr
  ret i32 %val
}
```

Listing 9: The `nonnull` attribute indicates that the pointer argument should never be `null`

2. **Metadata:** This is optional information about an instruction or global object, conveying additional hints to analysis and optimization passes and code generation. This information can be safely ignored by transformations without compromising correctness. In Listing 10, the `!range` metadata is used to indicate that the loaded value is guaranteed to be in the range $[0, 100)$. If the value is outside this range, an erroneous value is returned.

```
define i32 @foo(ptr %x) {
  %v   = load i32, ptr %x, !range !0
  %res = mul i32 %v, 2
  ret i32 %res
}
!0 = !{i32 0, i32 100}
```

Listing 10: The range metadata indicates that the loaded value must be in the range $[0, 100)$

3. **Intrinsics:** These are internal functions with semantics defined by LLVM. Intrinsics represent an extension mechanism to LLVM IR which does not require changing all of the transformation passes when adding a new language construct. In Listing 11, the `@llvm.memcpy` intrinsic copies 8 bytes from the memory location pointed to by `%p` to the memory location pointed to by `%q`.

```
define void @foo(ptr %p, ptr %q) {
  call void @llvm.memcpy.p0.p0.i64(ptr %q, ptr %p, i64 8, i1 false)
  ret void
}
```

Listing 11: The call to the `@llvm.memcpy` intrinsic copies 8 bytes

Choosing the right construct to encode a given piece of information minimizes memory usage and compile time overhead, as each type of construct has a distinct expressive power and performance cost. Attributes present the lowest expressive power, as these are only applied to function parameters and return values, but also constitute a lower cost to optimization passes. Metadata information increases the domain of attributes, as it can be attached to any instruction or global object. Although optimization passes can optionally ignore metadata, processing lots of metadata can slow down compilation. Intrinsics can convey information that the two other mechanisms cannot, but constitute the most expensive construct. As these are represented as opaque function calls in the IR, they often block optimizations.

3

Undefined Behavior

Contents

UB plays a crucial role in the design of the IR of an optimizing compiler, such as LLVM. It is used to convey to the optimizer that certain operations do not produce a well-defined result. This simplifies the compilation process and grants the compiler greater freedom to optimize, thereby improving the size and performance of the generated code.

In the context of this work, UB is particularly relevant as it models the semantics of low-level operations, such as memory loads, playing an important role in defining the semantics of the IR and how memory is managed. This thesis aims to extend LLVM with a new type that closely interacts with memory, demanding a thorough understanding of how UB is utilized within LLVM.

In Section 3.1, we explore the motivation behind incorporating UB into the IR of optimizing compilers. Then, in Section 3.2, we introduce the time traveling semantics of UB, present in LLVM IR. Finally, we delve into the use of UB within LLVM, in Section 3.3.

3.1 Motivation

Different architectures handle operations such as shift overflows, signed integer overflows, and unaligned memory accesses distinctly, serving as a motivating example for the role of UB. For example, x86 and ARM CPUs produce different results for shift operations where the shift amount is equal to or exceeds the bitwidth of the shifted value. On one hand, shifts of 32-bit values make use of the lower 5 bits to represent the shift amount on x86 CPUs, effectively shifting by the shift amount modulo 32. On the other hand, ARM uses 8 bits to represent the shift amount, causing the result to wrap around every 256 increments to the shift amount, as opposed to every 32. Therefore, operations such as `1 << 32` yield distinct results on both architectures, outputting `1` for X86 and `0` for ARM CPUs.

The IR of an optimizing compiler aims to be target-agnostic, ideally abstracting away machine-specific details. Incorporating target-specific semantics directly in the IR would overconstrain it and degrade performance on certain target architectures. For instance, defining the result of erroneous shift operations as `0` would require emitting an additional `xor` instruction on x86 CPUs to zero out the result register. Due to the semantic mismatches between different architectures, low-level programming languages such as C specify such operations as raising UB,[1] indicating that their result is deemed irrelevant, thereby improving both the performance and size of the generated code.

In another example, the C programming language introduces the concept of *indeterminate value*, "*either an unspecified value or a trap representation*", used to initialize objects without an explicit initializer.[2] Raising UB upon evaluating an uninitialized object allows the C programming language to remain performant, avoiding a default initialization mechanism, e.g., initializing variable `x` to `0`.

An optimizing compiler is only required to preserve the semantics of conformant programs, i.e. programs that do not raise UB, being allowed to optimize or eliminate UB-raising operations. This lifts responsibility from the compiler to the programmer by not requiring the former to preserve any particular behavior in the presence of such operations. It is instead the programmer's responsibility to ensure that programs are well-defined. In the extreme case, the compiler can optimize the whole program away. This is especially notorious in programming languages that make heavy use of UB, such as C and C++. In Listing 12, not all code paths of the program on the left initialize variable `x`. If variable `v` holds a value other than `4`, `x` is not initialized and is assigned the indeterminate value, raising UB when used as argument to the `printf` instruction. As the program on the left is non-conformant, the compiler can perform constant folding, as if variable `x` was initialized with the value `3` in all code paths, optimizing the program on the left into the program on the right.[3]

[1] *If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.* (C99 ISO Standard, 6.5.7)

[2] *The initializers of objects that have automatic storage duration [...] are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer).* (C99 ISO Standard, 6.8.3)

[3] https://godbolt.org/z/33o19canK

```c
int x;
if (v == 4) x = 3;
printf("%d\n", x);
```

```c
printf("%d\n", 3);
```

Listing 12: C program with uninitialized variable `x` (left), and optimized version (right) where `x` is replaced with `3`

Another goal of UB is to simplify the compilation process. Some program properties, such as the absence of out-of-bounds memory accesses, the non-dereference of null pointers or program termination, are generally undecidable, implying that compilers cannot in general prove such properties. However, by assuming a program is free of UB, compilers can, for instance, assume all indexing operations are valid, which drastically simplifies alias analysis and unlocks more optimization opportunities. More generally, UB allows compilers to generate highly optimized code without having to conservatively account for every possible corner case, which would severely worsen both compilation times, and the size and performance of the generated code.

3.2 Time Travel

In some programming languages, such as C++, UB has a time-travel semantics. Under the assumption that the source program is well-defined, the compiler is allowed to reorder, modify or eliminate code that precedes or calls an UB-raising operation. Particularly, in the presence of a non-conformant program, the optimizer is not required to preserve any of its observable behaviors, including I/O.

Listing 13 depicts a C++ function[4] which searches for an element matching the integer `v` in `array`. The program has an off-by-one error as, in the last iteration, variable `i` assumes the value of the size of the array, causing an out-of-bounds access on line 4. Therefore, either the loop terminates by returning `true` or the program raises UB. This allows the compiler to arbitrarily transform the program, as it is not required to preserve the semantics of ill-formed programs. Currently, the trunk version of `gcc` optimizes `array_contains` to always return `true`,[5] independently of the contents of `array` or the value of `v`.

```c
1  std::array<int, 4> array = {1, 2, 3, 4};
2  bool array_contains(int v) {
3    for (unsigned i = 0, e = array.size(); i <= e; ++i)
4      if (array[i] == v)
5        return true;
6    return false;
7  }
```

Listing 13: Time-travel semantics of UB allows `gcc` to optimize `array_contains` to always return `true`

In another example,[6] the function in Listing 14 returns a default value of `0` for a null pointer argument, and the dereferenced value for non-null pointers. However, the null check of `value_or_default` is elided

[4] Example adapted from https://devblogs.microsoft.com/oldnewthing/20140627-00/?p=633.
[5] https://godbolt.org/z/rdM1Yq8Y4
[6] Example adapted from https://devblogs.microsoft.com/oldnewthing/20140627-00/?p=633.

by the compiler, as the call to `printf` on line 2 dereferences the pointer without previously performing a null check. If the pointer argument is `nullptr`, the function raises UB. As the only UB-free path requires the pointer argument to be non-null, the function gets optimized to invariably return the dereferenced value. As `value_or_default` raises UB, the compiler employs time-travel semantics, optimizing function `foo` to always return `10`, as it constitutes the only valid, UB-free code path.[7] Thus, the optimized code does not execute the body of `value_or_default`, including the call to `printf`.

```
1  int value_or_default(int *p) {
2    printf("The value of *p is %d\n", *p);
3    return p ? *p : 0;
4  }
5
6  int foo(int done) {
7    if (!done)
8      return 10;
9    return value_or_default(nullptr);
10 }
```

Listing 14: Time travel semantics of UB causes the compiler to optimize `foo` to always return `10`

UB in LLVM also exhibits time-travel semantics. In Listing 15, the call to `%printf` either returns or raises UB, as specified by the `willreturn` attribute. Therefore, the program always raises UB, either in the call to `%printf`, or by reaching the `unreachable` instruction. The time-travel semantics of LLVM's UB allows it to optimize away the function call, transforming the function on the left into the one on the right.

```
define void @src() {
  call void @printf(...) willreturn
  unreachable
}
```

```
define void @tgt() {
  unreachable
}
```

Listing 15: Time travel semantics of LLVM's UB justifies the removal of the call to `printf`

3.3 Undefined Behavior in LLVM

LLVM IR is treated as a compilation target by many programming languages. Some of these make heavy use of UB, such as C and C++. To accurately model their semantics, LLVM IR incorporates two kinds of UB. The first kind, immediate UB, models serious errors that trap the CPU or corrupt memory, such as dividing by zero or dereferencing a null pointer. The other kind, deferred UB, models less serious errors that, nevertheless, produce unreliable results, such as signed integer overflow. Deferred UB is necessary to enable speculative execution, allowing UB operations to be hoisted out of loops. LLVM models the latter kind of UB through the `undef` and `poison` values, and a `freeze` instruction that stops their propagation.

[7] https://godbolt.org/z/991bGcsGE

3.3.1 Undef Value

The `undef` value models a non-deterministic value of a given type. Its purpose is illustrated by the C function[8] in Listing 16, where variable `x`, defined but not initialized, is immediately returned. Lowering `foo` to LLVM IR requires emitting a particular value for variable `x` to be initialized with. One possible approach is to assign a default value to uninitialized variables, as in the Java programming language, e.g., initializing `x` with `0`. However, as previously mentioned, this constitutes a performance penalty for some architectures. By contrast, using `undef` to model C's indeterminate value avoids this overhead.

```
int foo() {
  int x;
  return x;
}
```

```
define i32 @foo() {
  ret i32 undef
}
```

Listing 16: Program in C (left), returning an uninitialized variable, and the corresponding lowering to LLVM IR (right)

The `undef` value grants greater flexibility to optimization passes, as it can be refined to any value of the given type. Nonetheless, it presents a major limitation. In Listing 17, a desirable optimization is shown, transforming a multiplication by two into an equivalent addition. This optimization is invalid as, if variable `%i` assumes the `undef` value, each of its uses may yield a different value. On the left, `%res` represents an even value, while the addition on the right can return an odd value. One of the uses of `undef` could yield an even value, while the other yields an odd value, resulting in an odd sum. Currently, Alive2 reports this optimization as unsound,[9] exclusively due to the possibility of `%i` being `undef`.

```
%res = mul i32 %i, 2    |    %res = add i32 %i, %i
```

Listing 17: Duplicate uses of `undef` may yield distinct values, preventing the optimization of `mul` (left) into `add` (right)

Currently, `undef` values are used solely to represent loads from uninitialized memory, as most other use cases can instead be handled by the `poison` value, a slightly more powerful form of UB.

3.3.2 Poison Value

The inability to justify optimization involving signed integer overflows constitutes another limitation of the `undef` value. In Listing 18, a desirable transformation is shown, transforming $a + b > b$ into $b > 0$. Although seemingly correct, the optimization is unsound[10] due to signed integer overflow. If the addition overflows, the program on the left returns `false`, while the optimized program on the right returns `true`.

The `nsw` flag in `add` instructions causes the addition to return an erroneous value in the presence of signed integer overflow. Considering such erroneous value to be `undef` and assuming `%a` to be the largest 32-bit integer, the comparison of the previous example simplifies to `undef` $>$ `INT_MAX`. As, in this

[8] https://godbolt.org/z/h5Y7qYxqx
[9] https://alive2.llvm.org/ce/z/EGxdoG
[10] https://alive2.llvm.org/ce/z/AzRb2B

```
define i1 @src(i32 %a, i32 %b) {
  %add = add i32 %a, %b
  %cmp = icmp sgt i32 %add, %a
  ret i1 %cmp
}
```

```
define i1 @tgt(i32 %a, i32 %b) {
  %cmp = icmp sgt i32 %b, 0
  ret i1 %cmp
}
```

Listing 18: Unsound optimization, due to signed integer overflow

context, `undef` models any given value of the `i32` type, the comparison returns `false`, as, by definition, there is no `i32` value greater than `INT_MAX`.

The `poison` value addresses this limitation by modeling and propagating the result of erroneous operations. Returning `poison` in the presence of signed integer overflow yields a comparison of the form `poison` $<$ `INT_MAX`. This expression evaluates to `poison`, as this value propagates itself through the expression Directed Acyclic Graph (DAG). The `poison` value does not model a value of a particular type, making it strictly more powerful than `undef`, allowing it to be refined to any value, be it `undef` or a concrete value. The optimization is now justified as the result of the comparison can be refined to `true`.

This kind of value is also used to represent padding bits. These are unused bits inserted by the compiler, often to satisfy alignment requirements or to fill gaps within data structures. Accesses to padded values pose a problem, as `poison` bits taint the load result, i.e., if at least one of the loaded bits is `poison`, the loaded value is `poison`.

The presence of `poison` bits, either in the form of padding bits or uninitialized memory, also invalidates load widening optimizations. On some CPU architectures, it is often more efficient to widen or combine loads to align with the word size. For instance, if a CPU has a 32-bit word size, loading a 16-bit value from memory might typically involve reading the entire 32-bit word and extracting the relevant portion. However, if any of the remaining 16 bits is `poison`, the load instruction returns `poison`.

3.3.3 Freeze Instruction

Despite the increased expressive power of the `poison` value, its unrestricted propagation through the expression DAG inhibits some optimizations. In Listing 19, a desirable optimizations hoists the conditional statement out of the loop. However, assuming `i` to be negative, the optimization is unsound if `c` is assigned the `poison` value, as the loop in the program on the left in never executed, while the program on the right raises UB, when branching on a `poison` value on line 3.

This limitation was addressed by a recent proposal [1] introducing the `freeze` instruction, which stops the propagation of `undef` and `poison` values through the expression DAG by picking a non-deterministic value, such that all uses of the same `freeze` instruction observe the same value. If the operand to this instruction is neither `undef` nor `poison`, it is a no-op. Freezing the value of `c` on line 3 of the program on the right validates the previous optimization, as the program no longer branches on the `poison` value and raises UB.

```
1  void src(int i, int x, int c) {
2    for (; i >= 0; --i) {
3      if (c) { f(1 / x); }
4      else { f(x); }
5    }
6  }
```

```
1  void tgt(int i, int x, int c) {
2    // correct if we use `if (freeze(c))`
3    if (c) {
4      for (; i >= 0; --i) { f(1 / x); }
5    } else {
6      for (; i >= 0; --i) { f(x); }
7    }
8  }
```

Listing 19: Hoisting the conditional out of the loop is unsound, as `c` could hold `poison`, making the target program (right) raise UB when branching on this value, on line 3, when the source program (left) is well-defined. It is important to note that this example is presented in C for clarity and conciseness, as the corresponding LLVM IR would be considerably more convoluted.

A common misuse of the `freeze` instruction lies in duplicating the instruction under the assumption that each distinct `freeze` returns the same value. However, this is generally invalid, as each `freeze` instruction may select a different value. Only uses of a singular `freeze` instruction observe the same result. Therefore, the optimization in Listing 20 is invalid, as the value of `y`, on the right, is not guaranteed to be preserved across all iterations of the loop, unlike in the program on the left.

```
void src(int a, int b, int c) {
  int x = a / b;
  int y = freeze(x);
  while (--c) {
    f(y);
  }
}
```

```
void tgt(int a, int b, int c) {
  while (--c) {
    int x = a / b;
    int y = freeze(x);
    f(y);
  }
}
```

Listing 20: Distinct uses of `freeze` may observe different values, invalidating the optimization

4

Memory Models

Contents

A memory model determines how memory is abstracted, defining the semantics of memory-related operations. Some programming languages, such as Java, provide references that behave like pointers. However, Java programs are not allowed to build references to arbitrary byte offsets within an object. On the other hand, low-level languages like C and C++ allow arbitrary memory inspection and manipulation.

This thesis aims to extend LLVM's type system with a new type that closely interacts with memory. Therefore, it is crucial to understand how memory models constraint the representation and manipulation of memory. In Section 4.1, we introduce the notion of high-level optimization and identify the characteristics that a memory model must exhibit to support such transformations. Then, in Section 4.2, we explore common low-level idioms, which must also be supported by the memory model. Finally, in Sections 4.3 and 4.4, we explore the two most employed kinds of memory models, regarding their representation of pointers, highlighting their strengths and limitations.

4.1 High-Level Optimizations

High-level optimizations greatly contribute to the performance of the code generated by the compiler. These require the memory model to establish certain invariants, such as local exclusive ownership, preventing aliasing with unescaped local variables, and absence of unpredicted aliasing, often caused by the presence of integer-to-pointer casts. A memory model that enforces these invariants can greatly enhance the precision of alias analysis, setting up a good foundation for enabling high-level optimizations.

As a motivational example, consider Listing 21, where a C function allocates an integer on the heap, sets its value to `0` and prints it to the screen. Between the last two steps, there is a call to an unknown function `bar`, on line 3. A memory model that naively allows arbitrary memory manipulation is unable to justify a constant propagation optimization, where `x` is replaced with `0` in the call to `printf`. Covertly, `bar` could craft a pointer aliasing with `x`, via an integer-to-pointer cast, and overwrite its value after the assignment on line 2, despite the address of `x` never being escaped, that is, passed as argument and observed by `bar`. In the absence of aliasing guarantees from the memory model, the compiler must conservatively refrain from optimizing.

```
1  int *x = malloc(sizeof(int));
2  *x = 0;
3  bar();               // could craft a pointer aliasing with `x`
4  printf("%d\n", *x); // we want to replace `x` with `0`
```

Listing 21: Potential aliasing through an integer-to-pointer cast in `bar` prevents a constant propagation optimization

High-level optimizations are also hindered by address guessing, where the address of a variable is guessed by the program and used to access the corresponding object. This weakens the precision of alias analysis. The program in Listing 22 can either print `0` or `1`, depending on the behavior of the memory allocator. This prevents the compiler from replacing the argument to `printf` with a constant, as there is at least one execution where object `p` gets assigned the memory address `0x7ffdaabfe8f4`. The program observes this address via control-flow, changing the value of `*p` to `1`, and thereby invalidating the high-level optimization.

```
char *p = malloc(4); *p = 0;
int addr = 0x7ffdaabfe8f4;
if ((int)p == addr)           // `p` gets its address guessed via control-flow
  *(int*)addr = 1;
printf("%d\n", *p);           // can print 0 or 1
```

Listing 22: Address guessing hinders the constant propagation optimization

This problem can be addressed by exploiting the non-determinism of the memory allocator, using UB to rule out undesirable memory accesses. This prevents the program from guessing the address of a pointer that has not been directly observed, either via a pointer-to-integer cast or via control flow. In order to justify high-level optimizations, a memory model should, therefore, prevent address guessing.

4.2 Low-Level Idioms

A memory model should also strive to support low-level operations. These are particularly hostile to high-level optimizations, due to their exposure of the underlying memory layout, weakening alias analysis by enabling the program to trivially create unexpected aliasing. Throughout the remainder of this section, we explore some of the most relevant low-level idioms.

Low-level programming languages expose a mechanism to construct references to arbitrary memory regions. C and C++ allow casting of integers to pointers and vice-versa. The former type of casts exposes the address of a pointer, while the latter allows the construction of arbitrary memory references that can be used to access and modify memory. A pointer can also undergo a round trip of casts, being cast to an integer and back to a pointer. Although the C standard defines most conversion between pointers and integers as implementation-defined,[1] except for the integer constant `0` and `NULL` pointers,[2] it is expected of most implementations to yield a pointer with similar capabilities in a cast round trip.

C and C++, in particular, allow one-past-the-end pointer-to-integer casts, used to iterate over arrays in an idiomatic way and to implement C++'s `std::end` iterator. Although legal to take a reference to one element past the end of an object, dereferencing such a pointer raises UB.[3] Ideally, a memory model should successfully model this cast round trip, casting a pointer referencing the element that immediately follows the object to an integer, and casting the integer back to a pointer, such as in Listing 23. Pointer `p` holds the address of `x` and is incremented to point to the next integer in memory. Variable `i` is assigned the integral value representing the address of `p`, which is then reinterpreted as a pointer via an integer-to-pointer cast on line 4. In the following line, `q` is decremented to point to the address of `x`. The memory access on line 6 should be well-defined and equivalent to an access through pointer `p`, as defined on line 1.

```
1  int *p = &x;
2  p += 1;
3  unsigned long i = (unsigned long)p;
4  int *q = (int*) i;
5  q -= 1;
6  *q = 1;
```

Listing 23: Round trip casting of one-past-the-end pointers in C and C++

Additionally, a memory model must accurately define the semantics of pointer arithmetic and comparison operations. These constitute a major hazard due to their exposure of the underlying memory layout. Defining pointer subtraction as an address difference allows a program to compute offsets be-

[1] *An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined,(...)* (C17 ISO Standard, 6.3.2.3p5)

[2] *An integer constant expression with the value 0, or such an expression cast to type void*, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.* (C17 ISO Standard, 6.3.2.3p3)

[3] *If the result points one past the last element of the array object, it shall not be used as the operand of a unary * operator that is evaluated.* (C17 ISO Standard, 6.5.6.8)

tween different objects in memory, which can then be used to craft an aliasing pointer via an integer-to-pointer cast. The C programming language disallows such arithmetic operations,[4] raising UB when dereferencing a pointer resulting from subtracting two pointers referring to distinct allocations (excluding one-past-the-end pointers), as occurs on line 6 of Listing 24. Similarly, pointer comparisons with one-past-the-end pointers may reveal the underlying memory layout, exposing contiguous objects in memory.

```
1  int *p = malloc(4 * sizeof(int));
2  int *q = malloc(4 * sizeof(int));
3  ptrdiff_t offset = q - p;
4  int *r = p + offset;
5  if (r == q) {
6    *r = 1;
7  }
```

Listing 24: C program using an undefined pointer subtraction, as the two pointers refer to distinct allocations

Low-level bit manipulation also constitutes a widely used low-level idiom. Listing 25 shows the implementation of a red-black tree in the Linux kernel. Each node has two pointers, one for each of its children, and a color. The color is represented as an integer, which additionally encodes the pointer to the parent node. This implementation assumes a 4-byte alignment, implying that the two least significant bits of a pointer to a `rb_node` are `0`. The color of the node, red or black, is encoded in the least significant bit, while the pointer to the parent node is obtained by clearing the two least significant bits. A memory model should provide a concrete representation for pointers and allow their bit-level manipulation.

```
struct rb_node {
  unsigned long  __rb_parent_color;
  struct rb_node *rb_right;
  struct rb_node *rb_left;
};

#define __rb_parent(pc)    ((struct rb_node *)(pc & ~3))
#define __rb_color(pc)     ((pc) & 1)
```

Listing 25: Implementation of a red-black tree in the Linux kernel, using low-level bit manipulation

Low-level languages also support some form of type punning or transmutation. Rust exposes an `unsafe` function, `std::mem::transmute`, which reinterprets the bits of a value of one type as a value of another, equally sized type. Particularly, transmutations between integer and pointer values are supported. C++ provides `reinterpret_cast`, which serves a similar purpose.

Lastly, memory models should support copies of pointer values, either via `memcpy` or via a user-defined copy. The latter can be implemented by leveraging the unrestricted access to object representations presented by the `unsigned char` type,[5] both in C and C++, or by the `signed char`, `char`,

[4] *When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object(...)* (C17 ISO Standard, 6.5.6.9)

[5] *Values stored in non-bit-field objects of any other object type consist of n x `CHAR_BIT` bits, where `n` is the size of an object of*

and `std::byte`[6] types in C++. These represent individual raw bytes, having their value preserved across memory loads and stores. Such user-defined copies take advantage of strict aliasing rules, as character-typed pointers may alias any object type.[7] This allows pointers to be copied without circumventing alias analysis. In Rust, although no formal memory model is currently specified for the language, the `MaybeUninit<T>` type[8] serves a similar purpose by representing possibly uninitialized raw bytes.

4.3 Flat Memory Models

The simplest kind of memory model treats memory as a flat array of bytes, closely mirroring the behavior of physical hardware. These memory models, known as flat or concrete memory models, represent pointers as integers of equivalent bitwidth. This allows integers to serve as indices to the flat byte array represented by memory, mapping them directly to memory addresses. Casts between pointers and integers are modeled as no-ops, grating concrete memory models native support for low-level idioms, as pointers are solely represented by their addresses, which can be captured by integer values. Consequently, pointer arithmetic, comparisons, and bit-level operations map directly to their integer equivalents. Additionally, type punning between pointers and integers and copies of pointer values are trivially supported, since both types are treated interchangeably.

Despite their simplicity, flat memory models are usually not adopted by optimizing compilers, as they fail to justify simple high-level optimizations. In the following example, represented in Figure 4.1, a constant propagation optimization is desirable, replacing `q[0]` in the last instruction with `0`, as this value remains unmodified after its assignment on line 3. By modeling memory as a byte array, the concreteness of the flat memory model gives a definite semantic to out-of-bounds indexing. If `q` immediately proceeds `p` in memory, `p[4]` and `q[0]` refer to the same memory location. In other words, `p[4]` and `q[0]` alias, invalidating the optimization, as the assignment on line 4 modifies the value of `q[0]`. The behavior of the program is, therefore, dependent on the behavior of the memory allocator. The compiler must conservatively consider all possible memory configurations and refrain from optimizing.

4.4 Logical Memory Models

Logical memory models tackle the aliasing problem present in flat memory models by abstracting away the memory layout. Memory is modeled as an infinite set of disjoint logical blocks, each associated with

that type, in bytes. The value may be copied into an object of type `unsigned char [n]` (e.g., by `memcpy`); the resulting set of bytes is called the object representation of the value. (C99 ISO Standard, 6.2.6.1.4)

[6] *The underlying bytes making up the object can be copied into an array of `char`, `unsigned char`, or `std::byte`. If the content of that array is copied back into the object, the object shall subsequently hold its original value.* (C++23 ISO Standard, 6.8.1.2)

[7] *If a program attempts to access the stored value of an object through a glvalue whose type is not similar to one of the following types the behavior is undefined: (...) a `char`, `unsigned char`, or `std::byte` type.* (C++20 ISO Standard, 7.2.1.11)

[8] https://doc.rust-lang.org/beta/std/mem/union.MaybeUninit.html

```
1  char *p = malloc(4);
2  char *q = malloc(4);
3  q[0] = 0;
4  p[4] = 1;              // if p[4] == q[0], this assignment concerns q[0]
5  printf("%d\n", q[0]);  // if p[4] == q[0], replacing q[0] by 0 is incorrect
```

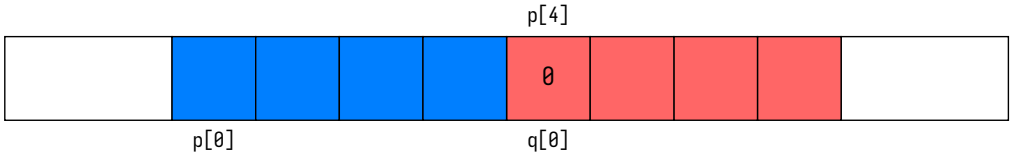


Figure 4.1: Aliasing in a flat memory model: if `q[0]` and `p[4]` alias, performing constant propagation is unsound

a unique allocation site. Pointers no longer refer to specific indices of the byte array, but rather to the allocation from which they are derived, disallowing programs from crafting a pointer with an arbitrary address and using it to overwrite memory referent to other allocations. Pointers are traced back to individual allocations through provenance metadata.

This provenance information is associated pointers, which are represented as a pair composed of a block identifier b_i, referent to the originating allocation, and an offset o_i within that block. Provenance is preserved by pointer arithmetic operations, as these only change the offset portion. This prevents unwanted aliasing and enables high-level optimizations, as pointers referring to distinct allocations are guaranteed not to alias.

Revisiting the previous example, reframed in Figure 4.2, under a logical memory model, the constant propagation optimization is now valid as pointers `p` and `q` refer to different allocations, originating from distinct calls to `malloc`. Supposing that the allocation of `p` returns a logical pointer (b_1, o_1), the allocation of `q`, referent to a different call to `malloc`, must return a logical pointer (b_2, o_2), such that $b_1 \neq b_2$. Since these pointers reference different logical blocks, they are guaranteed not to alias. As a result, the assignment of `p[4]` does not clobber the store to `q[0]`, allowing the compiler perform the optimization, replacing `q[0]` by `0` in the last instruction.

```
char *p = malloc(4);
char *q = malloc(4);
q[0] = 0;
p[4] = 1;
printf("%d\n", q[0]);
```

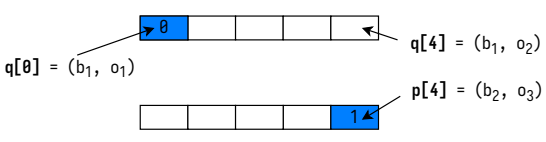


Figure 4.2: An example program in C (left) and a visual representation of its logical memory layout (right)

At runtime, there is no tracking or enforcement of memory accesses through provenance information, as it is an abstract concept, exclusively used by the compiler to justify optimizations and simplify the compilation process. It can rule out out-of-bounds accesses, simplifying alias analysis and thereby enabling more optimizations. In Listing 26, the store on line 3 raises UB, as the memory access is

out-of-bounds of object `p`, from which pointer `q` is derived from.

```
1  int *p = malloc(4 * sizeof(int));
2  int *q = p + 8;
3  *q = 1; // out-of-bounds access
```

Listing 26: Out-of-bounds memory access raises UB

With the addition of provenance to pointers, some low-level idioms are no longer trivially supported. Firstly, it is unclear what the result of inter-object pointer arithmetic should be. The subtraction of two pointers sharing the same provenance can be interpreted as the difference between their logical offsets. However, there is no clear semantics for the subtraction of pointers referent to different allocations. Returning the difference between their block identifiers is technically defined but semantically meaningless, while returning the difference of their concrete addresses would expose the underlying memory layout.

Secondly, pointer equality comparisons should behave consistently, either only comparing addresses, or including provenance when both addresses compare equal. Incorporating inconsistent semantics can be problematic. Listing 27 shows a value numbering optimization, in which the equality of `p + 4` and `q` apparently enables these expressions to be replaced interchangeably. Considering the comparison to not take provenance into account, a memory layout where `p` immediately precedes `q` will cause the program to take the conditional branch on line 3. While the store guarded by the conditional on the program on the left is well-defined, the corresponding store on the right raises UB, as `p + 4` points to one element past the end of the object it refers to, constituting an out-of-bounds access. It is worth noting that, as previously mentioned, a program is allowed to craft a pointer to one element past the end of an object, but disallowed from dereferencing such a pointer without the appropriate provenance.

```
1  char *p = malloc(4);
2  char *q = malloc(4);
3  if (p + 4 == q)
4    *q = 1;
```

```
1  char *p = malloc(4);
2  char *q = malloc(4);
3  if (p + 4 == q)
4    *(p + 4) = 1; // wrong to replace `q` by `p + 4`
```

Listing 27: Semantic ambiguity in pointer comparison operations invalidates an equality propagation optimization

Pointer copies also raise a concern, as provenance information should be accurately preserved or reconstructed during the copy process. The memory model should support both memory-related primitives, such as `memcpy` and `memmove`, as well as user-defined copies, relying on raw memory access types. Both should enable direct copying of memory contents without performing type punning, particularly avoiding reinterpreting pointers as integers. This reinterpretation causes pointer provenance to be discarded, hindering high-level optimizations.

Lastly, logical memory models lack the capability to natively model casts between pointers and integers. Representing memory as an infinite number of non-overlapping blocks introduces challenges when mapping finite-width integers to logical blocks. A purely logical memory model disallows integer-pointer casts, while other solutions, explored in Section 5.1, extend this purely logical model, granting

pointers a more concrete representation.

Such solutions fall into one of two flavors of logical memory models. The first kind, Provenance-not-via-Integers (PNVI), restrict provenance information to pointers while the second kind, Provenance-via-Integers (PVI), extend provenance to both pointers and integers.

4.4.1 PNVI Memory Models

The PNVI family of memory models does not track provenance through integers. Instead, it handles integer-to-pointer casts by attempting to reconstruct the provenance based on whether the memory address stores a valid object. Memarian et al. [18,19] define the following PNVI variants, each specifying when such reconstruction is permitted.

1. **PNVI plain**: This variant recreates provenance for any valid object at the specified memory address, yielding a pointer with valid provenance from an integer-to-pointer cast. It is permissive enough to allow almost arbitrary integer-to-pointer casts, creating unexpected aliasing even with unescaped local variables. This contests the validity of high-level optimizations.

2. **PNVI address-taken**: Provenance can be reconstructed for objects whose address has been taken. Even though the object's address was never converted to an integer or directly observed, its provenance can still be recreated. Taking the address of a local variable causes it to be susceptible to address guessing, as other functions could craft a pointer, via an integer-to-pointer cast, aliasing with the local object.

3. **PNVI exposed-address**: Provenance is only recreated for objects whose address has been escaped. An address is escaped if the pointer to an object was explicitly cast to an integer, via a pointer-to-integer cast, or if the bit-level representation of the pointer was read through a non-pointer type. This variant is strict enough to disallow local aliasing, enabling the justification of high-level optimizations. Nonetheless, round trip casts of one-past-the-end pointers are not supported, as these do not directly expose the object whose address immediately precedes them, preventing provenance reconstruction.

4. **PNVI exposed-address-user-disambiguation**: This variant extends PNVI exposed-address to also support round trip casts. If a pointer is cast to an integer and back to a pointer, this variant will correctly reestablish the provenance of the pointer.

4.4.2 PVI Memory Models

PVI memory models tackle the problem of mapping an integer value to a logical block by also extending provenance to integers, enabling no-op integer-to-pointer casts in logical memory models. However,

tracking integer provenance hinders many widely used integer optimizations.

Simple arithmetic operations yield integers with restricted provenance, disabling foldings such as `i + 0` into `i`. Additionally, equality propagation optimizations become unsound when tracking integer provenance. In Listing 28, the guard on line 5 is only taken if `v` and `w` compare equal. Despite their identical numerical value, memory accesses through both integers, when cast to pointers, are not equivalent. This is because both integers originate from pointers with mismatching provenance, as these are based on different objects. While the store through `w` on the left is well-defined, the store on the right raises UB, as `v` is out-of-bounds of the object `p`.

```
1  char *p = malloc(4);
2  char *q = malloc(4);
3  int v = (int)p + 4;
4  int w = (int)q;
5  if (v == w)
6    *(int*)w = 2;
```

```
1  char *p = malloc(4);
2  char *q = malloc(4);
3  int v = (int)p + 4;
4  int w = (int)q;
5  if (v == w)
6    *(int*)v = 2; // wrong to replace `w` by `v`
```

Listing 28: Granting provenance to integers disables simple integer optimizations

5

Related Work

Contents

While only a limited set of existing work directly addresses the problem of copying raw memory values, several works explore closely related problems. In this chapter, we review these contributions and highlight their relevance to our work.

In Section 5.1, we delve into some of the most prominent memory models for low-level programming languages, with a particular emphasis on those reconciling high-level optimizations with low-level idioms, such as pointer arithmetic and integer-pointer casts. These include LLVM's adopted memory model, as well as an informal, alternate solution to our proposal. We analyze and compare these memory models according to a set of criteria, pinpointing their advantages and limitations.

In Section 5.2, we explore a proposal extending Clang to fully support C's `restrict` keyword. This work extends LLVM's `load` instruction with pointer provenance metadata, possibly strengthening alias analysis, and enabling explicit copies of pointer values by extending `memcpy` intrinsic calls with annotations marking the locations of pointers within copied memory values.

5.1 Memory Models

Various memory models have been proposed [19–24], some aiming to reconcile high-level optimizations with low-level idioms. Most memory models justify high-level optimizations by attaching provenance to pointers, representing them as logical entities, as opposed to mere memory addresses. This section explores some of the most relevant memory models, highlighting their strengths and weaknesses according to the following criteria, and comparing them in Table 5.1.

1. **Memory cardinality:** Memory models offer either a finite or infinite representation of memory. Infinite memory models represent memory as an infinite set of disjoint memory blocks, offering an unbounded pointer type. Thus, memory can never be exhausted, as there are an infinite number of memory addresses. This justifies optimizations taking memory cardinality into account, such as adding new allocations and removing existing dead allocations, without requiring any additional reasoning. On the other hand, finite memory models resemble the physical memory of a machine, representing memory as a finite set of memory blocks. Thus, memory can be exhausted, requiring additional effort to justify transformations that demand additional memory, such as register allocation spilling, or optimizations that may change the observable behavior of the program, such as dead allocation removal. Removing an allocation may alter the program's observable behavior, as the source program may run out of memory while the target program may not. In practice, a memory model should address finite memory, as compilation targets physical machines with finite memory. Ideally, it should support low-level idioms, while handling out-of-memory behaviors in a way that justifies high-level optimizations without excessive additional reasoning.

2. **Support for high-level optimizations:** These greatly contribute to the performance of the generated code. A memory model should justify high-level optimizations by abstracting the underlying concrete memory model of the physical machine, which gives a defined semantics to out-of-bounds indexing, weakening alias analysis and ultimately undermining many optimization opportunities. Ideally, a memory model should prevent unwanted aliasing, simplifying alias analysis. It should also preserve common integer optimizations, such as equality propagation and range analysis.

3. **Support for integer/pointer casts:** Pointer-integer casts are a common low-level programming idiom, abundantly used in embedded programming. A memory model should support them without imposing side-effects on the memory layout, thereby enabling cast elimination and code motion optimizations. Furthermore, casting a pointer to an integer and casting it back to a pointer in a cast round trip should also be supported.

4. **Support for low-level pointer manipulation:** Low-level pointer manipulation is a widely used low-level idiom in C and C++ code, such as operating systems, device drivers, and embedded

Memory Model	Memory	High-Level Memory Optimizations	`int`/`ptr` Casts	Low-Level `ptr` Manipulation	No Address Guessing
CompCert	Infinite	✓			
Quasi-Concrete	Finite	✓	Effectful	✓	
Twin-Allocation	Finite	✓	✓	✓	✓
Two-Phase	Infinite/Finite	✓	✓	✓	✓
CompCertS	Finite	✓	✓		
Limited Type Punning	Finite	✓			

Table 5.1: Summary of various memory models: how memory is modeled, whether the model supports high-level memory optimizations, whether the model supports integer-pointer casts, whether the model supports low-level pointer manipulation, and whether address guessing is mitigated, unblocking optimizations.

systems. Therefore, a memory model should allow pointers to be manipulated as plain integers, enabling low-level programming patterns such as XOR doubly-linked lists and hash tables with pointer values as keys. Furthermore, comparison[1] and arithmetic operations[2] should be defined for one-past-the-end object pointers.

5. **Address guessing mitigation:** The correctness of high-level optimization depends on the absence of address guessing patterns, where a pointer address is guessed due to deterministic allocation patterns. This allows the creation of aliasing pointers via integer-to-pointer casts, even with a pointer that never got its address observed. The crafting of such pointers weakens the precision of alias analysis, which yields more conservative results, and prevents many high-level optimizations from being performed. Thus, a memory model should rule out address guessing.

Throughout the remainder of this section, we delve into each memory model in detail, evaluating their characteristics and implications based on the outlined criteria.

5.1.1 CompCert

CompCert [25, 26] is a formally verified compiler targeting a subset of the C programming language. Its infinite memory model tracks provenance through pointers in order to justify high-level optimizations. Pointers are represented as pairs (b, o), where b denotes a logical block identifier and o indicates an offset within that block. Distinct allocations return logical pointers with differing block identifiers, preventing unwanted aliasing. Pointer arithmetic operations exclusively modify the offset field.

In the following example, depicted in Figure 5.1, pointers `a` and `b` originate from different allocations,

[1] *Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.* (C17 ISO Standard, 6.5.9.p6)

[2] *[I]f the expression P points to the last element of an array object, the expression (P)+1 points one past the last element of the array object, and if the expression Q points one past the last element of an array object, the expression (Q)-1 points to the last element of the array object.* (C17 ISO Standard, 6.5.6.p8)

and therefore refer back to different block identifiers. Thus, these two pointers do not alias. However, pointer `c` is derived from pointer `a` and these two may alias, as they share the same block identifier.

```
int *a = malloc(4 * sizeof(int));
int *b = malloc(4 * sizeof(int));
*b = 1;
int *c = a + f(0);
printf("%d\n", *b);
```

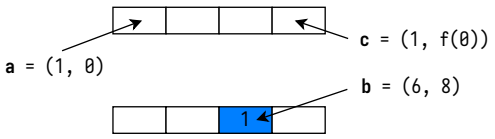


Figure 5.1: An example program in C (left) and a visual representation of its memory layout (right)

As an infinite memory model, optimizations such as dead allocation removal are always justified. However, this memory model breaks down in the presence of integer-pointer casts, as there is no clear mapping between pointers and physical memory addresses. Moreover, low-level pointer manipulation is not supported. As integer-to-pointer casts are not supported, address guessing fails to even be a concern.

5.1.2 Quasi-Concrete

The quasi-concrete [2] memory model gives a concrete semantics to arithmetic operations involving pointers that have been cast to integers. This is achieved by taking a hybrid approach, offering pointers both a concrete and a logical representation. A pointer returned from an allocation is logical, implying that it is not directly mapped to any physical memory address, preventing unwanted aliasing and enabling high-level optimizations. If that same pointer is never cast to an integer, the corresponding memory block remains logical throughout the whole program execution, as there is no need to expose its corresponding concrete address. In Listing 29, `p` is a logical pointer, as it never gets cast to an integer. This allows the value stored through `p` to be propagated across the call to `foo`, allowing the compiler to constant fold the argument to `printf`.

```
// `p` is a logical pointer
int *p = malloc(sizeof(int));
*p = 123;
foo();
printf("%d\n", *p);
```

```
// `p` is a logical pointer
int *p = malloc(sizeof(int));
*p = 123;
foo();
printf("%d\n", 123);
```

Listing 29: Constant propagation optimization allowed by the quasi-concrete memory model

In the absence of integer-to-pointer casts, the quasi-concrete memory model is equivalent to the memory model adopted by CompCert, enforcing the invariants expected by high-level optimizations. However, while CompCert's memory model panics in the presence of a pointer-to-integer cast, the quasi-concrete memory model allocates a block in concrete memory at cast time, mapping the logical pointer to the address of the newly allocated block. Once a pointer becomes concrete, it remains concrete throughout the rest of the program's execution, as once its address has been exposed, an aliasing

pointer can be crafted via an integer-to-pointer cast. Consequently, concrete pointers are susceptible to address guessing, even if only used locally.

This limitation is illustrated in Listing 30. The logical pointer `p` is guaranteed not to alias with any other pointer derived from a distinct object. As in the previous example, this enables the compiler to constant fold the argument to `printf` on line 5. However, the cast on line 7 turns `p` into a concrete pointer. Since its concrete address has now been observed, the call to `bar` may arbitrarily craft a pointer aliasing with `p` and change its value, despite `p` never being escaped. Thus, performing an identical constant propagation optimization, now on line 9, is disallowed.

```
1  // `p` is a logical pointer
2  int *p = malloc(sizeof(int));
3  *p = 123;
4  foo();
5  printf("%d\n", *p);
6  // `p` is now a concrete pointer
7  int a = (int)p;
8  bar();
9  printf("%d\n", *p);
```

```
1  // `p` is a logical pointer
2  int *p = malloc(sizeof(int));
3  *p = 123;
4  foo();
5  printf("%d\n", 123);
6  // `p` is now a concrete pointer
7  int a = (int)p;
8  bar();
9  printf("%d\n", *p);
```

Listing 30: Constant propagation optimization hindered by the presence of an integer-to-pointer cast

Offering pointers both a logical and concrete representation allows the quasi-concrete memory model to justify high-level optimizations that are unaffected by address guessing, while supporting pointer-integer casts and low-level pointer arithmetic. However, by assigning a concrete address at cast time, casts have the side-effect of allocating a block in concrete memory. As memory is finite, adding a pointer-to-integer cast to a program can result in an out-of-memory error. Furthermore, these casts cannot be removed, even if dead. In Listing 31, the cast on line 2 cannot be removed, as it allows `bar` to access the memory block pointed to by `p`, which would not be accessible if the cast was absent. Removing the cast causes the previously succeeding memory access performed by `bar` to fail, changing the program's observable behavior.

```
1  int *p = malloc(sizeof(int));
2  int a = (int)p; // cannot be removed, even though the cast is dead
3  bar();
```

Listing 31: The quasi-concrete memory model does not allow pointer-to-integer casts to be removed

Furthermore, this memory model is susceptible to address guessing. In Listing 32, replacing the argument in `printf` with the constant `10` is unsound, as `bar` could craft a pointer aliasing with `p` and change the value it points to. Despite `p` never getting its address observed, there is at least one program execution where it can successfully be guessed.

```
void foo() {
  int *p = malloc(sizeof(int));
  *p = 10;
  bar();
  printf("%d\n", *p);
}
```

Listing 32: Address guessing invalidates propagating the constant `10` to the `printf` call

5.1.3 Twin-Allocation

The twin-allocation [5] memory model adopts both logical pointers, used to justify high-level optimizations, and physical pointers, used to represent the result of integer-to-pointer casts and allow low-level pointer manipulation. It addresses the main limitation of the quasi-concrete memory model by assigning addresses at allocation time, rather than at cast time. This enables the removal of dead cast instructions, as these no longer produce a side-effect on the memory layout.

The address guessing problem is mitigated by changing allocation functions to reserve multiple memory blocks, rather than a single one. One of these blocks is non-deterministically chosen to be used, while the others become trapped, that is, their access raises UB. Revisiting the example in Listing 32, the optimization is now justified as, even if `bar` successfully guesses the address of `p` and crafts an aliasing pointer, there will be at least one execution where the accessed block was trapped.

This proposal also introduces a deferred bounds checking mechanism, enabling code motion optimizations. Out-of-bounds pointers can be created, only triggering UB when such a pointer is dereferenced. Listing 33 illustrates a scenario where, using an immediate bounds checking mechanism, the reordering of declarations can cause the optimized program to raise UB. Considering that `q` immediately follows `p` in the memory layout, the program on the left is allowed to deference `s` (which must be inbounds of the same object pointed to by `r`, as signified by `r +inb`), as it is based on `r`, which despite not being inbounds of `p`, is inbounds of `q`, encoding a valid memory access. However, if the allocation of `q` is performed after the declaration of `s`, its dereference raises UB. Despite `s` ending up pointing to a valid object, `r` is not in bounds of any valid object at the time of its declaration on line 2, on the right.

```
1  char *p = malloc(4);
2  char *q = malloc(4);
3  char *r = (char*)((int)p + 5);
4  char *s = r +inb 1;
5  *s = 0; // OK
```

```
1  char *p = malloc(4);
2  char *r = (char*)((int)p + 5);
3  char *s = r +inb 1;
4  char *q = malloc(4);
5  *s = 0; // UB
```

Listing 33: With immediate bounds checking, reordering declarations causes the dereference of `s` to raise UB

By adopting a deferred bounds checking mechanism, the optimization is justified. This is achieved by tracking the set of addresses that must be inbounds of the same object when a physical pointer is dereferenced. When performing inbounds pointer arithmetic, the base and resulting pointer addresses are stored, performing a valid memory access only when all of the stored addresses are within bounds

of the same object. Revisiting the previous example, reframed under Listing 34, addresses `0x15` and `0x16` must be within bounds of the same object when dereferencing `s`. Up to line 3, any dereference of `s` is invalid, as there is no valid object residing at that memory location. However, after the allocation of `q` on line 4, addresses `0x15` and `0x16` are in bounds of object `q`, allowing the memory access on line 5.

```
1  char *p = malloc(4);           // (val=0x10, obj=p)
2  char *r = (char*)((int)p + 5); // (val=0x15, obj=*)
3  char *s = r +inb 1;            // (val=0x16, obj=*, inb={0x15,0x16})
4  char *q = malloc(4);           // (val=0x14, obj=q)
5  *s = 0; // OK since 0x15 and 0x16 are inbounds of same object
```

Listing 34: Deferred bounds checking allows code motion optimizations by tracking the set of in bounds addresses

The twin-allocation memory model uses a finite representation of memory. Therefore, any transformations that may eliminate an out-of-memory condition are unsound. For instance, in edge cases where the original program runs out of memory but its optimized version does not, removing a dead allocation can alter the program's observable behavior. Therefore, such transformations cannot always be soundly justified. This is the memory model currently in use by LLVM.

5.1.4 Two-Phase

The limitations inherent to finite memory models are addressed by the two-phase [6] infinite/finite low-level memory model. All previous memory models, except for CompCert, handle finite memory, being unable to always justify optimizations that eliminate out-of-memory conditions, such as dead allocation removals. To circumvent this limitation, two separate compilation phases are proposed.

The first phase operates over an infinite memory model, with unbounded memory addresses in bijection with an unbounded integer type, enabling a round trip of casts to yield the original pointer. Under the infinite memory model, high-level optimizations are justified and, in particular, as allocations always succeed, dead allocation removals are always sound. In contrast, the second compilation phase is performed under a finite memory model, enabling subsequent lowering to machine code. A translation is performed between these two memory models, preserving the semantics of the program, with the introduction of halt points due to memory exhaustion.

This memory model constructs symbolic values to represent the result of low-level operations. A normalization function is used to concretize the individual bytes represented by a symbolic value across all possible memory layouts, derivable from the current state of the program. This allows each symbolic byte to represent a partially defined state, expanding the set of low-level operations that can be represented, including support for pointer values as hash keys and using the least significant bit of a pointer as a flag. Additionally, other mechanisms are used to define more concretization scenarios, such as entanglement, ensuring that two symbolic bytes stored in the same operation are concretized together.

Address guessing is prevented by using non-deterministic allocation functions. Under the infinite

memory model, there will always be an execution where a guessed block could be allocated somewhere else. This is equivalent to the approach followed by the twin-allocation memory model of allocating trapped blocks.

While the two-phase memory model addresses the limitations of finite memory models, it introduces additional complexity, as two separate compilation phases are required. Compiler developers need to reason about optimizations both under an infinite and finite memory model and, most crucially, need to prove that the mapping between these two is correct. Furthermore, the conversion between the infinite and finite memory models inserts bounds checks on pointer arithmetic operations, used to introduce halt points in the second phase, possibly causing a performance overhead.

5.1.5 CompCertS

An alternative approach to the previously presented memory models is explored by CompCertS [27,28], which extends CompCert's memory model by leveraging symbolic values to semantically define low-level operations, such as pointer arithmetic and bit-level pointer manipulation. In standard CompCert, expressions produced by such operations are undefined. CompCertS prevents the immediate evaluation of such expressions by constructing symbolic values that hold information about the legal values an expression can evaluate to. By delaying their evaluation, a defined semantics can be given to a strictly larger set of programs.

A symbolic value models a delayed computation and, therefore, cannot be used indefinitely. At some point, when a concrete value is needed, a function is used to normalize the symbolic value. This normalization function is parameterized by the current memory state, returning a value such that it evaluates identically across all concrete memory layouts derivable from the current memory state. Otherwise, the normalization is undefined.

This mechanism enables pointers to be cast to integers, and supports bitwise operations over the resulting integer, encapsulated in a symbolic value. However, some low-level programming idioms are still not supported, such as using pointer values as hash keys or using the result of a pointer comparison to determine a lock order in a concurrent environment. This stems from the inability of the symbolic value to represent a unique integer value across all possible memory layouts.

Address guessing also poses a problem, inhibiting some high-level optimizations. Furthermore, this memory model assumes that programs never exhaust memory. However, by treating memory in a finite way, program transformations must be proven to not increase the memory usage of the program, which constitutes a major limitation in practice.

5.1.6 Limited Type Punning

The original RFC[3] introducing the byte type to LLVM IR faced a lot of skepticism due to the unclear explanation of the underlying issue causing miscompilations and the substantial engineering effort required to introduce a new type of the IR. This sparked a discussion, leading to a proposal which does not demand the introduction of a new type.[4] The proposed memory model tackles the same issues as the byte type by extending all LLVM values with provenance metadata.

This memory model introduces the concepts of allocation and capability. The former refers to an address and an array of bytes, while the latter encodes the permission to access some region of memory. LLVM values are represented as an array of bits, with each of these bits having both an optional numerical value and capability. The numerical and capability values of an LLVM object are determined by the values of the bits composing the object. More precisely,

- **Numerical value:** If at least one of the bits of the object does not have a defined numerical value (its value is `poison`), the numerical value of the object is `poison`. Otherwise, its numerical value is given by concatenating the numerical values of the individual bits.
- **Capability:** The capability of an object is given by the union of all the capabilities of its bits.

While the numerical value of an object represents its data, its capability represents the object's provenance. Therefore, integers are defined as objects whose bits all have a defined numerical value, but no capabilities. Conversely, the bits composing a pointer also all have a defined numerical value, but are all required to share the same capability.

This memory model prevents type punning when copying pointers, as loads and stores simply move bits from and to memory, without introducing implicit casts. Furthermore, by modeling LLVM values as a collection of bits, loading memory data containing unspecified values does not taint the result, as each bit's numerical value is independent of the others.

Despite the distinction in capabilities between integer and pointer values, the absence of a new type in the IR requires that both of these can be represented by an integer type with the appropriate bitwidth. Therefore, integer optimizations must either be disabled for integer values that contain capabilities, or must distinguish values with identical numerical values, but differing capabilities. The memory model proposes that integer operations return `poison` if one of its arguments contains capability bits or if any of its bits does not have a defined numerical value, effectively disabling integer optimizations for pointers.

Furthermore, the use of capability bits in integer values prevents common low-level idioms. For instance, pointer arithmetic operations are not supported by the memory model, as adding a pointer value, with capabilities, to an integer value, without capabilities, returns `poison`. Even a simple transformation,

[3] https://lists.llvm.org/pipermail/llvm-dev/2021-June/150883.html
[4] https://discourse.llvm.org/t/a-memory-model-for-llvm-ir-supporting-limited-type-punning/61948

such as folding `p + 0` to `p`, where `p` is a pointer, is not allowed. This could be circumvented by casting the pointer to an integer, losing the capability information, performing the arithmetic operation, and casting the result back to a pointer. However, the proposal is unclear on how to restore capability metadata. For identical reasons, low-level pointer manipulation is disallowed, as bitwise operations over operands with distinct capabilities are not supported. Moreover, address guessing is not accounted for.

5.2 Restrict Patches

The `restrict` keyword is a type qualifier that can be used on pointer object declarations, encoding that "*all accesses to that object use, directly or indirectly, the value of that particular pointer*". This qualifier restricts pointer aliasing, enabling more optimizations, such as loop vectorization. In Listing 35, `dst` is a restrict pointer and, therefore, does not alias with `lhs`, nor `rhs`, as these are not based on `dst`, allowing the loop to be vectorized.

```
void madd(int *restrict dst, int *lhs, int *rhs) {
  for (int i = 0; i < 1024; ++i) {
    dst[i] = dst[i] + lhs[i] * rhs[i];
  }
}
```

Listing 35: Declaring `dst` as a `restrict` pointer allows the loop to be vectorized

It is worth noting that the `restrict` keyword acts as a promise made by the programmer. There are no static analysis or dynamic checks enforcing the correctness of these annotations. Thus, any access to an object pointed to by a restrict pointer `p` made through a pointer not based on `p` raises UB.

There is an ongoing effort[5] to fully support `restrict` in LLVM. The current implementation of Clang only processes restrict pointer function arguments, ignoring all other restrict pointer declarations. The full-restrict patches[6] aim to introduce support for local restrict pointer declarations and restrict member pointers. Furthermore, they aim to fix miscompilations,[7] observed when inlining functions with restrict pointer arguments, while enhancing the performance of the current implementation.

The patches introduce an optional argument to `load` and `store` instructions, used to associate provenance information with the pointer operand, referencing which objects it may alias with. In Listing 36, the loaded value depends on the memory layout of the program. However, even if `q` immediately precedes `p` in memory, the address of the loaded value is given by `p[1]`. Whilst the pointer address is the same, provenance must be taken into account. To that effect, the `ptr_provenance` operand, on the right, specifies that the pointer value is based on `q` if `p` and `q` are adjacent in memory, or based on `p` otherwise.

[5] https://discourse.llvm.org/t/full-restrict-support-in-llvm/53267, https://discourse.llvm.org/t/rfc-yet-another-llvm-restrict-support/87612

[6] https://reviews.llvm.org/D68484

[7] https://bugs.llvm.org/show_bug.cgi?id=39282, https://bugs.llvm.org/show_bug.cgi?id=39240

```
if ((int)(p + 4) == (int)q) {
  return q[0];
} else {
  return p[1];
}
```

```
%p.1 = getelementptr i32, ptr %p, i64 4
%p.1.addr = ptrtoint ptr %p.1 to i32
%q.addr = ptrtoint ptr %q to i32
%cmp = icmp eq i32 %p.1.addr, %q.addr
%prov = select i1 %cmp, ptr %q, ptr %p
%res = load i32, ptr %p.1, ptr_provenance %prov
```

Listing 36: Use of the `ptr_provenance` operand to specify the provenance of the loaded pointer

Although the `restrict` patches and the byte type tackle orthogonal problems, the `ptr_provenance` operand can be used to recover some of the alias analysis precision forfeited in the absence of a dedicated type to represent raw memory values by raising UB if the pointer argument to a load/store instruction is not based on the pointer referenced by the operand.

Furthermore, the patches introduce an intrinsic used to preserve provenance information when copying aggregate data containing pointer values. In Listing 37, the assignment on the left, is lowered as a call to the `@llvm.memcpy` intrinsic. In order to preserve provenance information, the call to the `@llvm.noalias.copy.guard` precedes the call to the other intrinsic, locating pointers in the memory block being copied. Metadata `!9` indicates that an aggregate type with size 8 is being copied, there is a pointer at offset 0 with size 8, and that there is a single pointer being copied. This mechanism grants additional context to optimizations lowering calls to `memcpy` to load/store pairs.

```
struct vec {
  int *restrict v;
};

void res(struct vec *v1,
         struct vec *v2) {
  *v1 = *v2;
}
```

```
define void @res(ptr %v1, ptr %v2) {
  %guard = call ptr @llvm.noalias.copy.guard.p0.p0(
    ptr %v2, ptr null, metadata !9
  )
  call void @llvm.memcpy.p0.p0.i64(
    ptr %v1, ptr %guard, i64 8, i1 false
  )
  ret void
}
!9 = !{i64 8, i64 0, i64 8, i64 1}
```

Listing 37: The `noalias.copy.guard` intrinsic locates pointers in the data being copied by the `memcpy` intrinsic

While the new IR constructs introduced by the full-restrict patches annotate `memcpy` calls with pointer location information and enable more precise alias analysis results, these mechanisms present some major limitations. Firstly, calls to the `@llvm.noalias.copy` intrinsic may block optimizations. Furthermore, not all calls to the `@llvm.memcpy` intrinsic are required to be guarded by the new intrinsic. Otherwise, optimization passes that emit `memcpy` calls would be required to specify the location of pointers being copied within the copied value, which is undecidable at compile time. Therefore, lowering `memcpy` calls to memory loads through integer types remains unsound.

Moreover, the proposal does not directly address the fundamental issues tackled by our proposal. The `ptr_provenance` operand to load/store instructions specifies the provenance of the pointer operand, but conveys no information about the loaded value. Therefore, loads through integer types continue to discard provenance information. Similarly, loads of padded values continue to taint the load result.

6

Supporting Raw Data Copies in the LLVM Intermediate Representation

Contents

In this chapter, we discuss our proposal of adding a new type capable of representing and manipulating raw memory values to LLVM IR. We highlight the changes and additions made to the IR and define the rules of the operational semantics for instructions that manipulate the byte type. A summary of the IR changes is presented at the end of the chapter.

6.1 Preliminaries

Without loss of generality, a simplified abstract machine model of LLVM IR is used, derived from the full model of Lee et al. [5]. The simplified model assumes two value types: integers and 64-bit pointers. Values of both types can also assume the `poison` value.

Fig. 6.1 shows the definitions of our model. Memory (Mem) is a partial function from block identifiers to blocks, and Block is a pair consisting of its size and contents, defined as a sequence of bytes. Each allocation generates a new block, initialized to `poison`. A pointer is represented as a pair consisting of a block identifier and an offset within that block. Pointer arithmetic operations preserve the block identifier, only modifying the offset.

The set of all possible values of an integer of bitwidth w is given by $[\![\mathbf{i}w]\!]$, which also includes the `poison` value. A bit is either a 1-bit integer or a pointer bit (represented as a pair of a pointer and an offset within the pointer's address). For example, when a pointer p is stored to memory, the abstract model does 64 stores of $(p, 0)$ up to $(p, 63)$, one for each bit of the pointer's address.

The register map (Reg) is a function from register names to their type and value. All registers are initialized to `poison`. The machine state is represented as a pair $(\mathrm{Reg}, \mathrm{Mem})$.

$$
\begin{array}{lcl}
\mathrm{Num}(w) & ::= & \{\, i \mid 0 \leq i < 2^w \,\} \\
\mathrm{BlockID} & ::= & \mathbb{N} \\
\mathrm{Mem} & ::= & \mathrm{BlockID} \rightharpoonup \mathrm{Block} \\
\mathrm{Block} & ::= & \{\, (n, c) \mid n \in \mathbb{N} \wedge c \in \mathrm{Byte}^n \,\} \\
\mathrm{Pointer} & ::= & \{\, (id, o) \mid id \in \mathrm{BlockID} \wedge o \in \mathrm{Num}(64) \,\} \\
[\![\mathbf{i}w]\!] & ::= & \mathrm{Num}(w) \uplus \{\, \mathbf{poison} \,\} \\
[\![\mathbf{ptr}]\!] & ::= & \mathrm{Pointer} \uplus \{\, \mathbf{poison} \,\} \\
\mathrm{Bit} & ::= & \{\, \mathrm{Int}(n) \mid n \in \{0, 1\} \,\} \uplus \{\, \mathrm{Ptr}(p, i) \mid p \in \mathrm{Pointer} \wedge (0 \leq i < 64) \,\} \uplus \{\, \mathbf{poison} \,\} \\
\mathrm{Byte} & ::= & \mathrm{Bit}^8 \\
\mathrm{Name} & ::= & \{\, \texttt{\%x}, \texttt{\%y}, \ldots \,\} \\
\mathrm{Reg} & ::= & \mathrm{Name} \to \{\, (ty, v) \mid v \in [\![ty]\!] \,\}
\end{array}
$$

Figure 6.1: Definitions. The set of all possible values of a type ty is given by $[\![ty]\!]$.

6.2 Byte Type

The byte type represents raw memory values, preserving provenance information and individual `poison` bits. It is a first-class single-value type, with the same size and alignment as the equivalently sized integer type. Memory loads through the byte type yield the value's raw representation, without introducing implicit casts. Both pointer and non-pointer values can be converted to the byte type.

Figure 6.2 defines auxiliary functions, used to define the conversion of bytes to values of other types

$$\begin{aligned} \mathbf{i}w\Downarrow v &= \lambda i.\, \mathsf{Int}(\mathrm{getbit}(v,i)) \\ \mathbf{ptr}\Downarrow p &= \lambda i.\, \mathsf{Ptr}(p,i) \end{aligned}$$

(a) Converting a value to a byte.

$$\mathbf{i}w\Uparrow b = \begin{cases} n & \text{if } \forall_{0\le i<w}\,.\, \mathsf{Int}(\mathsf{getbit}(n,i)) = b[i] \\ \mathbf{poison} & \text{otherwise} \end{cases}$$

(b) Converting a byte to an integer.

$$\mathbf{ptr}\Uparrow b = \begin{cases} p & \text{if } \forall_{0\le i<64}\,.\, \mathsf{Ptr}(p,i) = b[i] \\ \mathbf{poison} & \text{otherwise} \end{cases}$$

(c) Converting a byte to a pointer.

$$\mathsf{freeze}(b) = \begin{cases} b & \text{if } b = \mathsf{Ptr}(\_,\_) \vee b = \mathsf{Int}(\_) \\ b' & \text{otherwise} \\ & \text{such that } b' \text{ is a fresh bit} \end{cases}$$

$$\mathsf{CastToInt}(b) = \begin{cases} n & \text{if } \forall_{0\le i<\mathsf{len}(b)}\, \exists_{n'}\,.\, (\mathsf{Int}(n') = b[i] \vee (\exists_{j,p}\,.\, \mathsf{Ptr}(p,j) = b[i] \wedge n' = \mathsf{addr}(p,j))) \wedge \\ & \qquad\qquad \mathsf{getbit}(n,i) = n' \\ \mathbf{poison} & \text{otherwise} \end{cases}$$

$$\mathsf{CastToPtr}(b) = \begin{cases} p & \text{if } \forall_{0\le i<64}\,. \quad \mathsf{Ptr}(p,i) = b[i] \\ \mathsf{Ptr}(0,\mathsf{n}) & \text{if } \forall_{0\le i<64}\, \exists_{n'}\,.\, (\mathsf{Int}(n') = b[i] \vee (\exists_{j,p}\,.\, \mathsf{Ptr}(p,j) = b[i] \wedge n' = \mathsf{addr}(p,j))) \wedge \\ & \qquad\qquad \mathsf{getbit}(n,i) = n' \\ \mathbf{poison} & \text{otherwise} \end{cases}$$

Figure 6.2: Auxiliary functions used in defining the semantics of the byte type in LLVM IR. Let $\mathsf{len}(b)$ denote the number of bits in a byte b, $b[i]$ the i-th bit of b, $\mathsf{getbit}(n,i)$ the i-th bit of the number n in its binary representation, and $\mathsf{addr}(p,i)$ the i-th bit of the address in binary of pointer p.

$$\frac{(\iota = \text{“}r = \mathbf{bitcast}\ ty\ v\ \mathbf{to}\ \mathbf{b}w\text{”}) \qquad \text{BITCAST} \quad v' = ty\Downarrow R[v]}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto v'], M}$$

Figure 6.3: Rule of the operational semantics for the `bitcast` instruction.

and vice-versa. (a) shows functions for converting values into bytes. (b) and (c) define how to convert a byte into an integer and a pointer, respectively. If any bit does not match the target type, the result is `poison`. Freezing a bit is a no-op for non-`poison` values and produces a fresh bit otherwise. The CastToInt function forcefully converts a byte into an integer. It differs from $ty\Uparrow b$ because it coerces pointers to integers by extracting the corresponding bit from their address. Similarly, the CastToPtr function converts a byte into a pointer, coercing integers into pointers by returning a pointer with an address given by the concatenation of the binary value of the bits, and no provenance. Specifically, the returned pointer is $(0, addr)$, where $addr$ is the address value, and 0 denotes the null block (of zero size).

Figure 6.3 shows the rule of the operational semantics for the `bitcast` instruction. This instruction was extended to accept the byte type as a destination type, allowing values of other types to be converted to the byte type. The cast result is a byte value holding the individual bits of the cast operand. In

particular, `poison` bits are preserved at the bit-level, not tainting the cast result. Both loads of padded values and `bitcast` conversions between vector and scalar byte types preserve the raw byte value. In Listing 38, on the left, the `bitcast` from a vector to a scalar byte type (`b64`) does not taint the cast result. On the right, the cast returns `poison`, caused by spreading the `poison` lane.

```
%v = load <2 x b32>, ptr %ptr
%c = bitcast <2 x b32> %v to b64
```

```
%v = load <2 x i32>, ptr %ptr    ; <i32 poison, i32 42>
%c = bitcast <2 x i32> %v to i64 ; i64 poison
```

Listing 38: A `bitcast` from a vector to a scalar byte (left) does not taint the result, unlike with integer types (right)

By representing both pointer and non-pointer values and preserving individual `poison` bits, the byte type enables the implementation of a user-defined `memcpy` in the IR, as shown in Listing 39. The presented implementation differs from the current lowering performed by LLVM on lines 12 and 14, where the memory value is loaded and stored through the byte type, rather than through an integer type. In a similar manner, a native implementation of `memmove` can be achieved.

```
1  define ptr @my_memcpy(ptr %dst, ptr %src, i64 %n) {
2  entry:
3    br label %for.cond
4
5  for.cond:
6    %i = phi i64 [ 0, %entry ], [ %inc, %for.body ]
7    %cmp = icmp ult i64 %i, %n
8    br i1 %cmp, label %for.body, label %for.end
9
10 for.body:
11   %arrayidx = getelementptr inbounds b8, ptr %src, i64 %i
12   %byte = load b8, ptr %arrayidx
13   %arrayidx1 = getelementptr inbounds b8, ptr %dst, i64 %i
14   store b8 %byte, ptr %arrayidx1
15   %inc = add i64 %i, 1
16   br label %for.cond
17
18 for.end:
19   ret ptr %dst
20 }
```

Listing 39: Native implementation of a user-defined `memcpy` in the LLVM IR

The `freeze` instruction is also extended to work on byte operands, with freezing applied on a per-bit basis. The rule of operational semantics for this instruction is shown in Figure 6.4.

$$\frac{(\iota = \text{“}r = \textbf{freeze}\ \textbf{b}w\ v\text{”}) \quad \text{FREEZE} \quad v' = (\text{freeze}(R[v][0]), \ldots, \text{freeze}(R[v][w-1]))}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto v'], M}$$

Figure 6.4: Rule of the operational semantics for the `freeze` instruction.

6.3 Bytecast Instruction

Byte values can be reinterpreted as values of other types through the new `bytecast` instruction. The cast source and destination types must have the same bitwidth. The cast is available in two variants, either allowing or disallowing type punning. The new instruction adheres to the following semantics, formally specified in Figure 6.5.

1. A non-`exact bytecast` is used to reinterpret a byte value as a value of any other type, allowing type punning. More precisely,
   - If the type of all bits underlying the byte value matches the cast destination, it is a no-op.
   - Otherwise, the cast operand undergoes a conversion to the destination type, converting pointer bits to their address representation and non-pointer bits to a pointer with no provenance. This pointer can be used in operations that query its address, such as comparisons, but cannot be dereferenced.
2. An `exact bytecast` returns a non-`poison` value if the type of all bits in the byte value and the cast destination type are either both pointer or non-pointer types. More specifically,
   - If the type of all bits underlying the byte value matches the cast destination, it is a no-op.
   - Otherwise, the result is `poison`, preventing coercions between pointer and non-pointer values.

$$\frac{(\iota = \text{“}r = \textbf{bytecast b}w\ v\ \textbf{to i}w\text{”}) \quad \text{BYTECAST-INT} \quad v' = \mathsf{CastToInt}(R[v])}{R, M \xhookrightarrow{\iota} R[r \mapsto v'], M} \qquad \frac{(\iota = \text{“}r = \textbf{bytecast b}64\ v\ \textbf{to ptr}\text{”}) \quad \text{BYTECAST-PTR} \quad v' = \mathsf{CastToPtr}(R[v])}{R, M \xhookrightarrow{\iota} R[r \mapsto v'], M}$$

$$\frac{(\iota = \text{“}r = \textbf{bytecast exact b}w\ v\ \textbf{to}\ ty\text{”}) \quad \text{BYTECAST-EXACT} \quad v' = ty \Uparrow R[v]}{R, M \xhookrightarrow{\iota} R[r \mapsto v'], M}$$

Figure 6.5: Rules of the operational semantics for the `bytecast` instruction.

The `exact` variant the `bytecast` mimics the memory reinterpretation of a value, as if it had been stored to memory and loaded back through the cast destination type. This is aligned with the semantics of the `bitcast` instruction, which “*is done as if the value had been stored to memory and read back as [the destination type]*”. In Listing 40, a value is loaded from memory and reinterpreted as an integer, via the `bytecast` instruction. The `exact bytecast` returns a valid integer value only if an integral value is stored in memory, preventing type punning. The second `bytecast` converts the raw memory value to an integer. If a pointer is stored in memory, the cast yields an integer representing the pointer address.

```
%b = load b8, ptr %p
%c1 = bytecast exact b8 %b to i8 ; returns poison if a pointer is stored in memory
%c2 = bytecast b8 %b to i8       ; performs type punning
```

Listing 40: LLVM IR program using the `bytecast` instruction to reinterpret a raw memory value as an integer

6.4 Byte Constants

LLVM defines constants of primitive types, including integer and floating-point constants. Our proposal introduces byte constants to the IR, which are equivalent to the corresponding integer constants. Listing 41 shows two equivalent LLVM IR programs storing scalar, vector, and array constants to memory. The function on the left uses byte constants, while the one on the right uses equivalent integer constants.

```
store b32 1, ptr %p
store <4 x b8> splat (b8 0), ptr %p
store [2 x b64] [b64 1, b64 2], ptr %p
```

```
store i32 1, ptr %p
store <4 x i8> splat (i8 0), ptr %p
store [2 x i64] [i64 1, i64 2], ptr %p
```

Listing 41: LLVM IR functions storing byte constants (left) and integer constants (right) to memory

Byte constants were introduced to enable the initialization of global variables of raw memory access types. Listing 42 shows the definition and declaration of a C `struct`, containing an `int` and `char` fields. The corresponding lowering to LLVM IR is shown on the right.

```
struct s {
 int i;
 char c;
} g = { 42, 'C' };
```

```
@g = global
   { i32, b8, [3 x i8] }
   { i32 42, b8 67, [3 x i8] zeroinitializer }
```

Listing 42: Lowering of global variables in a C program to LLVM IR

6.5 Bitwise and Arithmetic Operations

Byte values can be truncated using the `trunc` instruction. Extending this instruction to byte-typed operands enables the extraction of smaller-typed values from a byte, narrowing a wider byte value to a specific width. In Listing 43, on the left, the byte value `%b` is truncated to a `b8` value. Casting the byte to an integer and truncating the resulting value, as performed on the right, is not equivalent to directly truncating the byte value, as the `bytecast` could taint the cast result if any of the bits of `%b` were `poison`.

Due to the cumbersome semantics of performing arithmetic on provenance-aware values, arithmetic operations on the byte type are disallowed. Bitwise binary operations are also disabled, with the exception of the logical shift right operator, enabling the extraction of smaller byte values at an offset within a wider byte. In Listing 44, a byte value is shifted and truncated, extracting the third most significant byte from the original value. The operational semantics for both instructions are shown in Figure 6.6.

```
%r = trunc b32 %b to b8
```

```
%c = bytecast exact b32 %b to i32
%t = trunc i32 %c to i8
%r = bitcast i8 %t to b8
```

Listing 43: Correct (left) and wrong (right) truncation of a raw byte value using the `trunc` instruction

```
%shift = lshr b32 %x, 16
%trunc = trunc b32 %shift to b8
%cast = bytecast exact b8 to i8
```

Listing 44: Extraction of the third byte in a 32-bit byte value using the `lshr` and `trunc` instructions

$$(\iota = \text{“}r = \textbf{trunc b}w\ v\ \textbf{to b}w_2\text{”})$$
TRUNC
$$\frac{v' = (R[v][0], \ldots, R[v][w_2 - 1])}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto v'], M}$$

$$(\iota = \text{“}r = \textbf{lshr b}w\ a,\ b\text{”})$$
SHIFT-RIGHT
$$\frac{b' = \mathbf{i}w \Uparrow R[b] \qquad b' \neq \textbf{poison} \qquad 0 \leq b' < w \qquad v' = (R[v][b'], \ldots, R[v][w-1], \mathsf{Int}(0), \ldots, \mathsf{Int}(0))}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto v'], M}$$

Figure 6.6: Rules of the operational semantics for the `trunc` and `lshr` instructions.

6.6 Clang Type Lowerings

With the addition of the byte to LLVM IR, Clang's raw memory access types can now be lowered to the byte type, as depicted in Listing 45.

```
void foo(
  unsigned char arg1,
  char          arg2,
  signed char   arg3,
  std::byte     arg4
);
```

```
void @foo(
  b8 zeroext %arg1,
  b8 signext %arg2,
  b8 signext %arg3,
  b8 zeroext %arg4
)
```

Listing 45: Lowering of C and C++'s raw memory access types to the byte type

As previously mentioned, arithmetic operations on the byte type are disallowed. Therefore, Clang now emits `bytecast` instructions, converting bytes to integers where integral values were previously expected. Listing 46 depicts an example function in C, adding two `char` values. The previous and new lowering to LLVM IR are shown, respectively, in the middle and on the right.

```
char f(char c) {
  return c + 1;
}
```

```
define i8 @f(i8 %c) {
  %i32 = sext i8 %c to i32
  %add = add nsw i32 %i32, 1
  %t8  = trunc i32 %add to i8
  ret i8 %t8
}
```

```
define b8 @f(b8 %c) {
  %i8   = bytecast b8 %c to i8
  %i32  = sext i8 %i8 to i32
  %add  = add nsw i32 %i32, 1
  %t8   = trunc i32 %add to i8
  %byte = bitcast i8 %t8 to b8
  ret b8 %byte
}
```

Listing 46: Function in C (left) and respective old (middle) and new (right) lowering to LLVM IR

6.7 Summary

In summary, the byte type contributes with the following changes/additions to the IR:

1. **Raw memory representation:** Optimization passes can use the byte type to represent raw memory data in registers, avoiding the introduction of implicit casts and treating both pointer and non-pointer values uniformly.

2. **Bit-level `poison` representation:** The byte type provides the necessary granularity to represent individual `poison` bits, providing greater flexibility than the corresponding integer types, which either have a fully-defined value or are tainted by `poison` bits.

3. **Byte constants:** Byte constants were introduced to support the initialization of global variables. These are strictly equivalent to their integer counterparts.

4. `bitcast` **instruction:** This instruction allows conversions from other primitive types to equivalently-sized byte types. Casts between vector and scalar byte types do not taint the cast result in the presence of `poison` lanes, as occurs with integer types.

5. `bytecast` **instruction**: This instruction enables the conversion of byte values to other primitive types. The standard version of the cast performs type punning, reinterpreting pointers as integers and vice-versa. The `exact` variant of the cast disallows type punning by returning `poison` if the type of the value represented by the byte does not match the cast destination type.

6. `trunc` **and** `lshr` **instructions**: The `trunc` and `lshr` instructions accept byte operands, behaving identically to their integer counterparts.

7

Implementation

Contents

In the previous chapter, we delved into the theoretical aspects of supporting raw data copies in LLVM IR. In this chapter, we explore the implementation details of the byte type in LLVM/Clang,[1] and Alive2.[2] In Section 7.1, we enumerate the optimizations that were fixed with the introduction of the new type, as well as some of the newly implemented optimizations. We also describe the changes made to Clang's code generation. Finally, in Section 7.2, we provide a high-level overview of the implementation of the byte type in Alive2.

7.1 LLVM/Clang

The implementation of the byte type in LLVM and Clang spans approximately 2.6k lines of C++ code. The LLVM core libraries were extended with the new type and instruction, and several transformation

[1] GitHub Repository (LLVM): https://github.com/pedroclobo/llvm-project/tree/byte-type
[2] GitHub Repository (Alive2): https://github.com/pedroclobo/alive2/tree/byte-type

passes were modified. Fixing existing optimizations and introducing new ones required modifying 83 regression tests, totaling 5.9k changed lines. Lowering Clang's raw memory access types to the byte type disrupted code generation. Fixing it required inserting missing `bytecast` instructions and updating a total of 1.8k Clang regression tests, affecting 381k lines of test code (mostly automated replacements of expected output from `i8` to `b8`).

7.1.1 Fixed Optimizations

The following unsound optimizations were fixed with the byte type:

1. **Lowering of `memcpy` and `memmove` to load/store pairs:** The InstCombine pass performs peephole optimizations, replacing a small set of instructions with a more performant, usually smaller equivalent set of instructions. This optimization pass commonly lowers calls to `memcpy` and `memmove` to integer load/store pairs.[3] These lowerings are unsound as loads through integer types do not preserve pointers, nor padded values. Both lowerings have been reworked to use a byte load/store pair, as exemplified in Listing 47, preserving the raw memory value by not introducing implicit casts or tainting the loaded value.

```
call void @llvm.memcpy(ptr %dst, ptr %src, i64 8)
call void @llvm.memmove(ptr %dst, ptr %src, i64 8)
```

```
%l = load b64, ptr %src
store b64 %l, ptr %dst
```

Listing 47: New lowering of `memcpy` and `memmove` calls to byte load/store pairs by InstCombine

   The SROA optimization pass breaks up `alloca` instructions of aggregate types into individual allocations, ideally one for each member of the structured type. Additionally, it promotes `alloca` instructions to SSA registers. This pass performed a similar transformation to InstCombine, lowering `memcpy` calls to integer load/store pairs.[4] Similarly, this optimization was reworked to use byte load/store pairs, as depicted in Listing 48.

```
%m = alloca i8
call void @llvm.memcpy(ptr %m, ptr %p, i32 1)
call void @llvm.memcpy(ptr %p, ptr %m, i32 1)
```

```
%b = load b8, ptr %p
store b8 %b, ptr %p
```

Listing 48: New lowering of `memcpy` calls to byte load/store pairs by SROA

2. **Lowering of `memcmp` to load/sub pairs:** InstCombine lowers calls to `memcmp` to integer loads, followed by a subtraction comparing the two memory values.[5] As discussed before, this lowering is unsound. Loading the two memory values as bytes is insufficient as comparisons between these are undefined. This avoids overloading the IR by not supporting comparisons between pointers and provenance-unaware values. Thus, the type punning variant of the `bytecast` is used, forcefully

[3] https://alive2.llvm.org/ce/z/2YnwP_
[4] https://alive2.llvm.org/ce/z/ywpadX
[5] https://alive2.llvm.org/ce/z/bMvEsq

converting pointer values into their integer representation. The two values, then converted to integers, can be compared as previously. Listing 49 depicts the old and new lowerings of a `memcmp` of 1 byte.

```
define i32 @my_memcmp(ptr %p, ptr %q) {
  %lhsc = load i8, ptr %p
  %lhsv = zext i8 %lhsc to i32
  %rhsc = load i8, ptr %q
  %rhsv = zext i8 %rhsc to i32
  %res  = sub i32 %lhsv, %rhsv
  ret i32 %res
}
```

```
define i32 @my_memcmp(ptr %p, ptr %q) {
  %lhsb = load b8, ptr %p
  %lhsc = bytecast b8 %lhsb to i8
  %lhsv = zext i8 %lhsc to i32
  %rhsb = load b8, ptr %q
  %rhsc = bytecast b8 %rhsb to i8
  %rhsv = zext i8 %rhsc to i32
  %res  = sub i32 %lhsv, %rhsv
  ret i32 %res
}
```

Listing 49: Previous (left) and new (right) lowering of a `memcmp` call comparing 1 byte

3. **Load widening:** LLVM commonly widens memory loads when lowering `memcmp` calls.[6] The previously presented lowering falls short in the presence of such optimizations. Whilst using a larger byte type to load the memory value preserves its raw value, the `bytecast` to an integer type yields `poison` if any of the loaded bits are `poison`. This is problematic as the remaining bits added by the widened load could assume any value or even be uninitialized. Thus, in the presence of load widening optimizations, the following lowering, depicted in Listing 50, is performed. The `freeze` instructions freeze each individual bit of the byte values, being a no-op for non-`poison` values and producing a fresh bit otherwise. This prevents the `bytecast` to an integer type from tainting the cast result and returning `poison`.

```
%res = call i32 @memcmp(ptr %x, ptr %y, i64 2)
```

```
%lx  = load b16, ptr %x
%ly  = load b16, ptr %y
%fx  = freeze b16 %lx
%fy  = freeze b16 %ly
%bx  = bytecast b16 %fx to i16
%by  = bytecast b16 %fy to i16
%zx  = zext i16 %bx to i32
%zy  = zext i16 %by to i32
%res = sub i32 %zx, %zy
```

Listing 50: Lowering of `memcmp` calls, supporting load widening

4. **Value coercions:** Some optimization passes, such as GVN, perform unsound value coercions. GVN, standing for global value numbering, is a transformation pass performing common subexpression elimination, that is, identifying and replacing computationally equivalent expressions with a canonical value. Currently, a class of optimization where a pointer load is coerced to a non-pointer value or a non-pointer load is coerced to a pointer value is reported as unsound by Alive2.[7] Listing 51 shows a simplified example. The original code snippet is shown on the left, with the

[6] https://alive2.llvm.org/ce/z/92OrW_
[7] https://github.com/llvm/llvm-project/issues/124461

code optimized by GVN in the middle. The transformation is unsound, as if a pointer is stored in memory, the load through an integer type returns `poison`, tainting the value of `%b`. On the right, the raw memory value is loaded through the byte type, avoiding the implicit type punning that occurs when loading a memory value through an integer type. The `bytecast` instructions are then used to coerce the raw memory value to an integer and to a pointer.

```
%a = load i64, ptr %p
%b = load ptr, ptr %p
```

```
%a = load i64, ptr %p
%b = inttoptr i64 %a to ptr
```

```
%v = load b64, ptr %p
%a = bytecast b64 %v to i64
%b = bytecast b64 %v to ptr
```

Listing 51: Original program (left), unsound optimization by GVN (middle), and fixed optimization (right)

7.1.2 New Optimizations

We used the byte type to implement the following new optimizations. These improve performance by reducing the number of IR instructions and enabling existing optimizations.

1. **Bytecast elimination:** InstCombine performs cast elimination, removing or simplifying redundant type casts. The `exact` version of the `bytecast` instruction is suitable for cast elimination, as the `poison` value is returned if the cast source and destination types do not match. The optimizer can refine the `poison` value to any other value. In Listing 52, two cast round-trips are simplified. The first function reinterprets the byte value as a `float`, and casts it back to a byte. If the value represented by `%b` is a pointer, the cast in the first function returns `poison`, which is then refined to the original value of `%b`. It is worth noting that the type punning variant of the cast could not be eliminated. If `%b` encapsulated a pointer value, the cast would change the underlying type of the byte value. If `%c` was then cast back to a `float`, through a `bytecast exact` instruction, eliminating the first cast would make the subsequent cast return `poison`, and not a defined `float` value. In the second function, an integer is reinterpreted as a byte and then reinterpreted as an integer, returning the original value.

```
define b32 @src_float(b32 %b) {
  %f = bytecast exact b32 %b to float
  %c = bitcast float %f to b32
  ret b32 %c
}

define i8 @src_int(i8 %i) {
  %b = bitcast i8 %i to b8
  %c = bytecast b8 %b to i8
  ret i8 %c
}
```

```
define b32 @tgt_float(b32 %b) {
  ret b32 %b
}

define i8 @tgt_int(i8 %i) {
  ret i8 %i
}
```

Listing 52: Elimination of cast round-trip pairs including a `bytecast`

2. **Load/Bytecast combining:** A `load` through the byte type followed by an `exact bytecast` returns a defined value only if the type of the loaded value matches the cast destination type. Therefore, pairs of these instructions can be combined into a single load through the cast destination type, as in Listing 53. This optimization is restricted to instruction pairs with a single use, in order to avoid increasing the number of instructions in the optimized IR. Combining a load with the type punning variant of the `bytecast` would be unsound, as the memory value would be punned to the cast destination type, while the combined load would return `poison`.

```
%b = load b8, ptr %p
%c = bytecast exact b8 %b to i8
```

```
%c = load i8, ptr %p
```

Listing 53: Combining of load and `bytecast` pairs with a single use

3. **Store forwarding:** Store forwarding optimizations are commonly performed on loaded raw memory values. Listing 54 shows one such optimization, where a byte value is reinterpreted as an integer through the `bytecast` instruction.

```
%a = alloca b8
store b8 %b, ptr %a
%v = load i8, ptr %a
```

```
%v = bytecast exact b8 %b to i8
```

Listing 54: Store forwarding optimization, reinterpreting a byte as an integer

Additional store forwarding optimizations are enabled by the `lshr` and `trunc` instructions. In Listing 55, the least significant byte is extracted. Performing an `exact bytecast` to an integer, followed by a `trunc` to the `i8` type would be unsound, as if any of the unobserved bits of the byte value were `poison`, the cast would return `poison`, thereby invalidating the transformation.

```
%a = alloca b32
store b32 %b, ptr %a
%v = load b8, ptr %a
```

```
%v = trunc b32 %b to b8
```

Listing 55: Store forwarding optimization, using the `trunc` instruction

The optimization shown in Listing 56 is a generalization of the previous one, offsetting the raw memory value before truncating it.

```
%a = alloca b32
%gep = getelementptr i8, ptr %a, i64 2
store b32 %b, ptr %a
%v = load i8, ptr %gep
```

```
%shift = lshr b32 %b, 16
%trunc = trunc b32 %shift to b8
%v     = bytecast exact b8 %trunc to i8
```

Listing 56: Store forwarding optimization, using the `trunc` and `lshr` instructions

4. **Bytecast SLP vectorization:** The Superword-Level Parallelism (SLP) vectorizer combines independent scalar instructions into vector instructions. In Listing 57, a sequence of scalar `load` and

`bytecast` instructions is vectorized. The `insertelement` instruction inserts its second operand into the vector given by the first operand, at the position specified by the third operand, producing a new vector with the updated element.

```
%p1 = getelementptr inbounds b8, ptr %p0, i64 1
%b0 = load b8, ptr %p0
%b1 = load b8, ptr %p1
%x0 = bytecast b8 %b0 to i8
%x1 = bytecast b8 %b1 to i8
%v0 = insertelement <2 x i8> poison, i8 %x0, i32 0
%r  = insertelement <2 x i8> %v0, i8 %x1, i32 1
```

```
%l = load <2 x b8>, ptr %p0
%r = bytecast <2 x b8> %l to <2 x i8>
ret <2 x i8> %r
```

Listing 57: SLP vectorization of scalar `load` and `bytecast` instructions

5. **Replace byte with integer constants in store instructions:** The byte type is used to represent a raw memory value. While byte values are opaque, treating pointer and non-pointer values uniformly, byte constants expose the byte's underlying type. As byte constants are strictly equivalent to the corresponding integer constant, the latter can replace the former in `store` instructions, as depicted in Listing 58. This unlocks more optimization opportunities by leveraging existing transformations.

```
store b8 1, ptr %p
store <2 x b8> <b8 1, b8 2>, ptr %p
```

```
store i8 1, ptr %p
store <2 x i8> <i8 1, i8 2>, ptr %p
```

Listing 58: Replacement of byte constants by the equivalent integer constants in `store` instructions

6. **Constant Folding:** Casts involving constant expressions are prime candidates for constant folding. In Listing 59, a `bytecast` from a vector of bytes (`<2 x b8>`) to a vector of integers (`<2 x i8>`) is folded into a constant vector and a constant splat vector of type `<16 x i8>` is reinterpreted through a `bitcast` as a vector of two `b64` values (`<2 x b64>`).

```
%v1 = bytecast <2 x b8> <b8 1, b8 2> to <2 x i8>
%v2 = bitcast <16 x i8> splat (i8 2) to <2 x b64>
call void @use(<2 x i8> %v1)
call void @use(<2 x b64> %v2)
```

```
call void @use(<2 x i8> <i8 1, i8 2>)
call void @use(<2 x b64> splat
               (b64 144680345676153346))
```

Listing 59: Constant folding of `bytecast` and `bitcast` instructions

7.1.3 IR Generation

Lowering Clang's raw memory access types to the byte type disrupted IR generation, as byte values were introduced where integer values were previously expected. The changes made to Clang consist of introducing missing `bytecast` instructions, preserving original IR generation functionality.

An as example, Listing 60 shows a C function that takes a `char` argument and returns its pre-incremented value, along with its corresponding previous and current lowerings to LLVM IR. Previously,

Clang lowered the `char` type to the `i8` integer type, allowing arithmetic operations such as addition to be performed directly on the function argument. With a byte-typed argument, Clang must now emit a `bytecast` instruction, coercing the byte to an integer before performing the addition. Additionally, a `bitcast` instruction should also be emitted, reinterpreting the addition result as a raw byte.

```
char f(char c) {
  return ++c;
}
```

```
define i8 @f(i8 %c) {
  %add = add i8 %c, 1
  ret i8 %add
}
```

```
define b8 @f(b8 %c) {
  %i8   = bytecast b8 %c to i8
  %add  = add i8 %i8, 1
  %byte = bitcast i8 %add to b8
  ret b8 %byte
}
```

Listing 60: C function (left), and respective previous (middle) and current (right) lowering to LLVM IR

Similarly, IR generation for certain target-specific builtins was fixed by introducing missing `bytecast` instructions. Some backends provide a large number of such builtins, exposing target-specific operations. The IR generation for each of these operations is typically manually defined. Fixing the IR generation for these builtin operations constitutes most of the changes made to Clang.

Other changes to code generation were also implemented. For example, stores of character-typed constants are lowered by Clang to integer stores, as shown in Listing 61. Using integer constants rather than byte constants unlocks more optimization opportunities early in the pipeline by leveraging already existing optimizations.

```
void c(char *p) {
  *p = 1;
}

void uc(unsigned char *p) {
  *p = 1;
}
```

```
define void @c(ptr %p) {
  store i8 1, ptr %p
  ret void
}

define void @uc(ptr %p) {
  store i8 1, ptr %p
  ret void
}
```

Listing 61: Lowering of stores of character-typed constants in C (left) to integer constant stores in LLVM IR (right)

7.2 Alive2

Alive2 is a bounded translation validation tool for LLVM IR, fully automated through the use of an SMT solver. The addition of the byte type to Alive2, comprising approximately 600 lines of C++ code, enables the automatic validation of optimizations involving the new type.

7.2.1 Byte Type and Bytecast Implementation

In Alive2, values of any given type are represented by a `StateValue` entity, which encapsulates an SMT bit-vector and tracks `poison` value information. The byte type is implemented on top of the existing

representation of raw memory. The `Byte` class models an individual raw memory byte. A value of the byte type is expressed as a `StateValue` whose bit-vector is formed by concatenating the SMT representations of the individual bytes it comprises.

Similarly, the `bytecast` instruction builds on existing functionality operating on raw memory bytes. Alive2 provides functions to convert a `StateValue` into a collection of `Byte` objects (`valueToBytes`) and to reconstruct a `StateValue` from such a collection (`bytesToValue`). The `bytecast` instruction is implemented using these two functions: `bytesToValue(valueToBytes(v, from_type), to_type)`.

The non-`exact` version of the cast, which performs type punning, was implemented using the existing assembly mode. This mode is used to verify transformations following assembly semantics, which allow type punning. When operating in assembly mode, pointer bytes are coerced to integers and vice-versa without, introducing `poison` bits.

7.2.2 Pre-Processor

Alive2 has a pre-processing step, gathering information about the source and target programs. The pre-processor exposes a set of global variables, which aim to reduce the complexity of SMT queries and thereby speed up the verification process. One of such variables is `bits_byte`, corresponding to the number of bits in the byte representation used by Alive2. The pre-processor walks through memory instructions, such as stores and `memcpy`, and `bitcast` instructions, registering the smallest addressable memory size. In Listing 62, the function on the left stores an 8-bit integer to memory. Therefore, its smallest addressable memory size is 8. However, the function on the right only stores a 32-bit integer value to memory, presenting a smallest addressable memory size of 32. The pre-processing was altered to also consider `bytecast` instructions to compute the smallest addressable size.

```
; bits_byte = 8
define void @f(i8 %v, ptr %p) {
  store i8 %v, ptr %p
  ret void
}
```

```
; bits_byte = 32
define void @f(i32 %v, ptr %p) {
  store i32 %v, ptr %p
  ret void
}
```

Listing 62: The `bits_byte` variable in Alive2 registers the smallest addressable memory size

Due to the close interaction of the byte type with memory, some of the pre-precessor optimizations are unsound in the presence of the new type. One such optimization occurs when the pre-processor does not identify any pointer stores in the program and defines memory refinement as an object comparison between `Byte` entities. This causes the transformation shown in Listing 63 to be incorrectly reported as unsound. This problem was addressed by altering the pre-processor to look for a `bytecast exact` to a pointer type followed by a `bitcast` back to a byte. In the presence of such a cast round-trip, the refinement of pointer bytes no longer performs a strict object comparison, but rather a full refinement check, which is more costly.

```
define b64 @src(b64 %b) {
  %1 = bytecast b64 exact %b to ptr
  %2 = bitcast ptr %1 to b64
  ret b64 %2
}
```

```
define b64 @tgt(b64 %b) {
  ret b64 %b
}
```

Listing 63: Optimization incorrectly reported as unsound by Alive2

In another example, the transformation shown in Listing 64 was initially reported as sound. However, as previously discussed, the cast pair cannot be eliminated as the non-`exact` version of the `bytecast` performs type punning, potentially reinterpreting pointers as integers. Therefore, the cast pair is not guaranteed to yield the original value. The pre-processor applied an optimization, only using pointer bytes only if either the source or target programs loaded/stored a pointer to/from memory. This eliminates the need to distinguish between pointer and non-pointer bytes, reducing the size of SMT queries. However, a non-`exact` `bytecast` instruction provides a new way of observing pointer values. Therefore, this particular transformation can only be successfully verified by considering the byte value to hold a pointer. The pre-processor was changed to look for non-`exact` `bytecast` instructions or a `bytecast` to a pointer type.

```
define b64 @src(b64 %b) {
  %1 = bytecast b64 %b to ptr
  %2 = bitcast ptr %1 to b64
  ret b64 %2
}
```

```
define b64 @tgt(b64 %b) {
  ret b64 %b
}
```

Listing 64: Optimization incorrectly report as sound by Alive2

8

Evaluation

Contents

In this chapter, we evaluate the implemented solution. In Sections 8.1 and 8.2, we list the benchmark programs and describe the testing environment. Then, in Section 8.3, we present the evaluation results, with respect to run-time performance, binary size, and other compilation metrics, including peak memory usage, compilation time, and the total number of LLVM IR instructions. In Section 8.4, we provide a detailed analysis on how some of the most significant binary size regressions were addressed, identifying their root cause and solution. Finally, in Section 8.5, we highlight the correctness contributions of the byte type to LLVM by identifying tests within the LLVM test suite and optimizations in the compilation of C/C++ programs that were fixed following the introduction of the new type. We also assess the performance impact of introducing the byte type to Alive2 by comparing the optimization verification times of these programs with those of the upstream version.

8.1 Benchmarks

The implementation was benchmarked using the Phoronix Test Suite,[1] from which a set of 20 C/C++ programs were selected. These programs, listed in Table 8.1, cover a wide range of application domains, code size, and performance characteristics, including both CPU and memory-bound workloads. In total, 5 million lines of code (LoC) were compiled, with `llvm` being the largest program with over 2 million lines of code. Many of the presented benchmarks have more than one performance test. For example, the `graphics-magick` benchmark assesses image processing performance across a range of operations, such as color space conversion, noise addition, enhancement, resizing, rotation, sharpening, and special effects like swirl. Overall, 64 performance tests were run.

No	Benchmark	Category	Measurement Scale	kLoC
1	aircrack-ng-1.3.0	Security	K/s	67
2	botan-1.6.0	Security	MB/s	148
3	build-llvm-1.5.0	Compiler	Seconds	2,209
4	compress-7zip-1.11.0	Compression	MIPS	247
5	compress-zstd-1.6.0	Compression	MB/s	90
6	draco-1.6.1	Texture Processing	Seconds	50
7	encode-flac-1.8.1	Audio Compression	Seconds	63
8	espeak-1.7.0	Speech Synthesizer	Seconds	45
9	graphics-magick-2.2.0	Image Processing	Iterations/Second	267
10	john-the-ripper-1.8.0	Security	Real Crypts/Second	315
11	jpegxl-1.6.0	Image Compression	MP/s	137
12	luajit-1.1.0	Compiler	Mflops	69
13	mafft-1.6.2	HPC	Seconds	94
14	ngspice-1.0.0	Circuit Simulation	Seconds	528
15	openssl-3.0.1	Security	MB/s	598
16	rnnoise-1.1.0	Audio Processing	Seconds	147
17	simdjson-2.0.1	Parallel Processing	GB/s	74
18	sqlite-speedtest-1.0.1	Database	Seconds	251
19	tjbench-1.2.0	Parallel Processing	MP/s	57
20	z3-1.0.1	SMT Solver	Seconds	512

Table 8.1: Benchmarked programs, along with their respective category, measurement scale, and approximate number of lines of C/C++ code (kLoC). The version number refers to the Phoronix benchmark version.

8.2 Setup

The experiments were conducted on a server with an AMD EPYC 9455 (Turin) 48-Core CPU, 128 GB of RAM, running Ubuntu 24.04.3 LTS. To minimize environmental noise, we disabled unnecessary system

[1] https://www.phoronix-test-suite.com

services, ran all benchmarks in single-threaded mode, and used the `taskset` and `nice` utilities to pin programs to a single CPU core and assign them maximum scheduler priority. We also disabled address space layout randomization (ASLR), and relied on the statistical methods built into the Phoronix Test Suite to ensure that the relative standard deviation of the results remained below 2%. Each program was compiled three times with the `-O3` optimization flag, and the average compile time between these runs was recorded. Each benchmark was executed at least three times. If the result variance exceeded 2%, additional runs were conducted to reduce it below that threshold.

8.3 Results

The plots in Figure 8.1 show the compile time (top-left), run-time performance (top-middle), peak memory usage (top-right), binary size (bottom-left) and LLVM IR instruction count (bottom-right) results of evaluating the 20 benchmark programs using the three prototype variants:

1. Enabling the byte type in LLVM IR and using it in Clang to compile `char` variables;
2. The same as variant (1), but with the optimizations over byte values enabled.
3. Variant (2) plus folding non-`exact` `bytecast` onto `load` instructions.

The third variant is included to explore the impact of changing the semantics of `load` instructions to allow type punning between pointers and integers, enabling the folding depicted in Listing 65.

```
%b = load b8, ptr %p
%i = bytecast b8 %b to i8
```

```
%i = load i8, ptr %p
```

Listing 65: Combining of `load` and `bytecast` pairs

8.3.1 Run-Time Performance

As shown in the top-middle plot in Figure 8.1, introducing the byte type causes performance regressions of up to 6.4% (average of 1.6%), with 5 benchmarks exceeding the 2% threshold. These slowdowns arise because adding a new IR instruction requires updating cost models and instruction matchers. Without these changes, many optimizations are overly conservative and ignore code with the new instruction. With our new optimizations, the worst slowdown drops to 4.4%, the average to 0.8%, and only two benchmarks exceed the 2% threshold. Enabling type punning in`load` instructions reduces the average slowdown further to 0.2%.

The `simdjson` benchmark is the only case that retains a noticeable slowdown. This occurs because LLVM's jump threading optimization can no longer merge certain basic blocks as its range analysis exclusively operates on integers. Extending this analysis to support the byte type is left for future work.

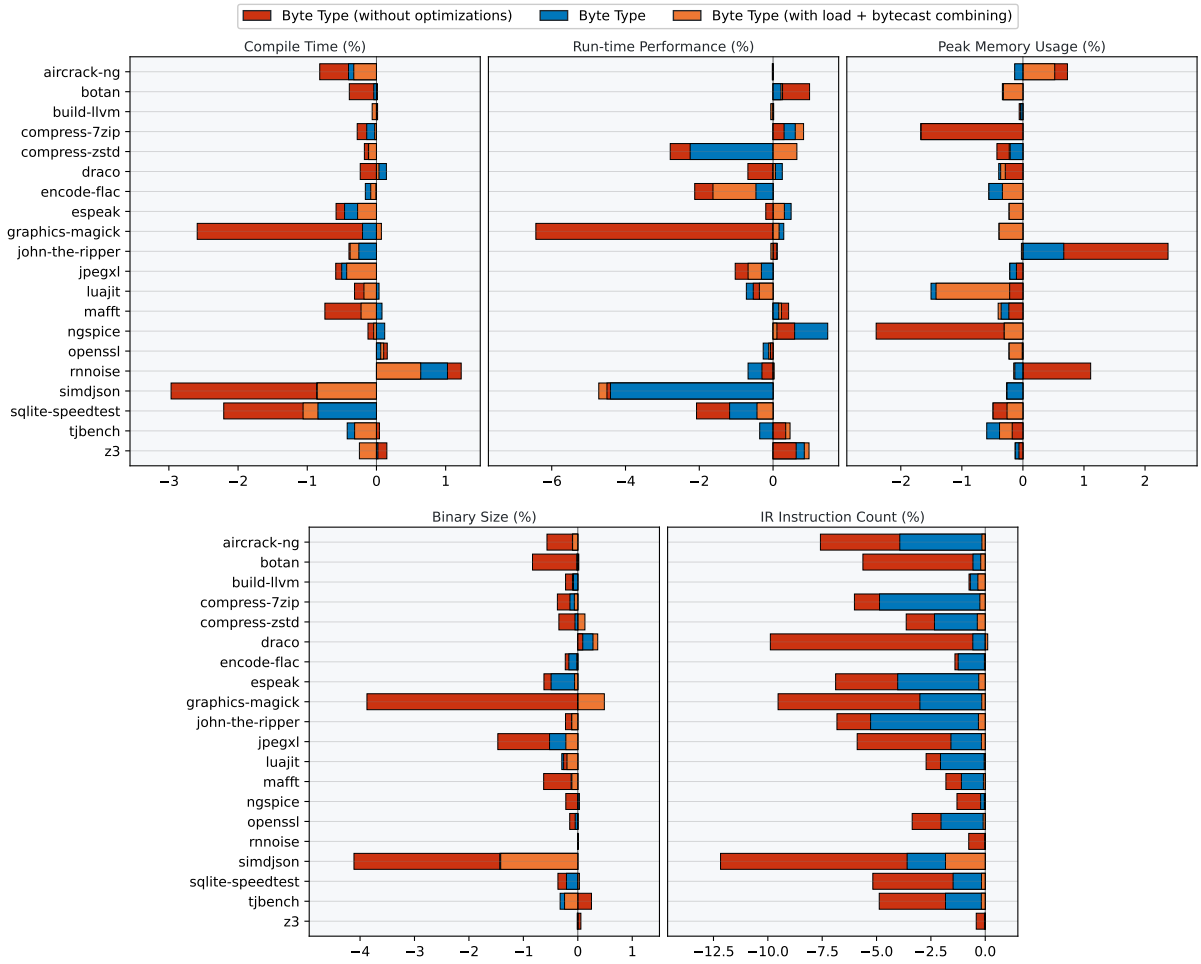


Figure 8.1: Benchmark results: compile time, run-time performance, peak compilation memory usage, binary size, and total number of LLVM IR instructions. Results are reported as the improvement percentage relative to the baseline. Positive values indicate improvements (e.g., faster compilation time or smaller binaries). We show 3 bars: enabling the byte type, enabling it with optimizations, and enabling it with folding `bytecast` onto `load`.

8.3.2 Binary Size

In Figure 8.1, the bottom-left plot shows that enabling the byte type has minimal impact on binary size. With optimizations enabled, the average increase is only 0.2% (0.7% without optimizations, and 0.1% with load type punning).

As examples of the impact of our optimizations, adjusting the loop vectorizer's cost model to account for the byte type made it vectorize fewer loops, which eliminated binary size regressions caused by over-vectorization: 2.7% in `pbzip2`, 1.5% in `GraphicsMagick`, and 1.1% in `tjbench`. Enhancing the loop-idiom recognizer to detect `strlen`-like patterns involving the byte type fixed a 1% code-size regression in `espeak`. Finally, updating SimplifyCFG's cost model for the `bytecast` instruction enabled additional basic blocks to be merged in `aircrack-ng`, which in turn unlocked further optimizations and reduced

binary size by 1%.

To better understand the impact of our prototype on code size, we analyzed individual functions rather than only the overall binary size, which can mask localized differences. Figure 8.2 presents two histograms per benchmark: one showing the distribution of function sizes, and another showing size differences between the baseline and each prototype variant. Across all benchmarks, the differences are centered around zero, as expected, since our changes should not affect the generated assembly. Minor variations are typically due to different register allocation decisions: because the allocator is greedy and the problem NP-hard, small perturbations in the IR can lead to large differences in the final code. However, some benchmarks exhibit outliers with size differences of up to a few kilobytes. Most cases we investigated and fixed were due to cost model adjustments that had not yet been done. Most remaining discrepancies are likely due to the same reason. A more detailed analysis of some of the underlying causes and solutions to the most significant binary size regressions is performed in Section 8.4.

8.3.3 Compilation Metrics

We also measured several compilation metrics, including compilation time, peak memory usage, and the total number of LLVM IR instructions. These are important metrics given the substantial resources required to compile large programs.

Optimizations for the byte type improve all metrics, reducing the average compilation slowdown from 0.5% to 0.15%. This improvement is expected, as fewer IR instructions (the average instruction count increase drops from 4.8% to 2.0%) reduce memory usage (overhead reduced from 0.4% to 0.2%), which in turn lowers cache misses and compilation time. Allowing type punning in `load` instructions decreases the IR size further, reducing the instruction count overhead to 0.2%.

The optimizations we implemented reduce the number of `bytecast` instructions substantially, from an average of 2.5% to 1.6% of all instructions. With type punning, this ratio is reduced to 0.1%.

8.4 Binary Size Impact Analysis

The diagnosed local variations in binary size can be attributed to two primary factors: (1) differing decisions made the vectorization cost model in our implementation, (2) and the byte type blocking certain optimizations, which perform bitwise operations over raw memory values, detect and transform specific loop idioms, and simplify code in other ways, thereby enabling other optimizations. The cost model plays a key role in loop vectorization by selecting the optimal vectorization and unroll factors. These choices significantly impact the size of the generated code, particularly due to the effects of loop unrolling. Generally, the optimizations observed to have been blocked would reduce code size.

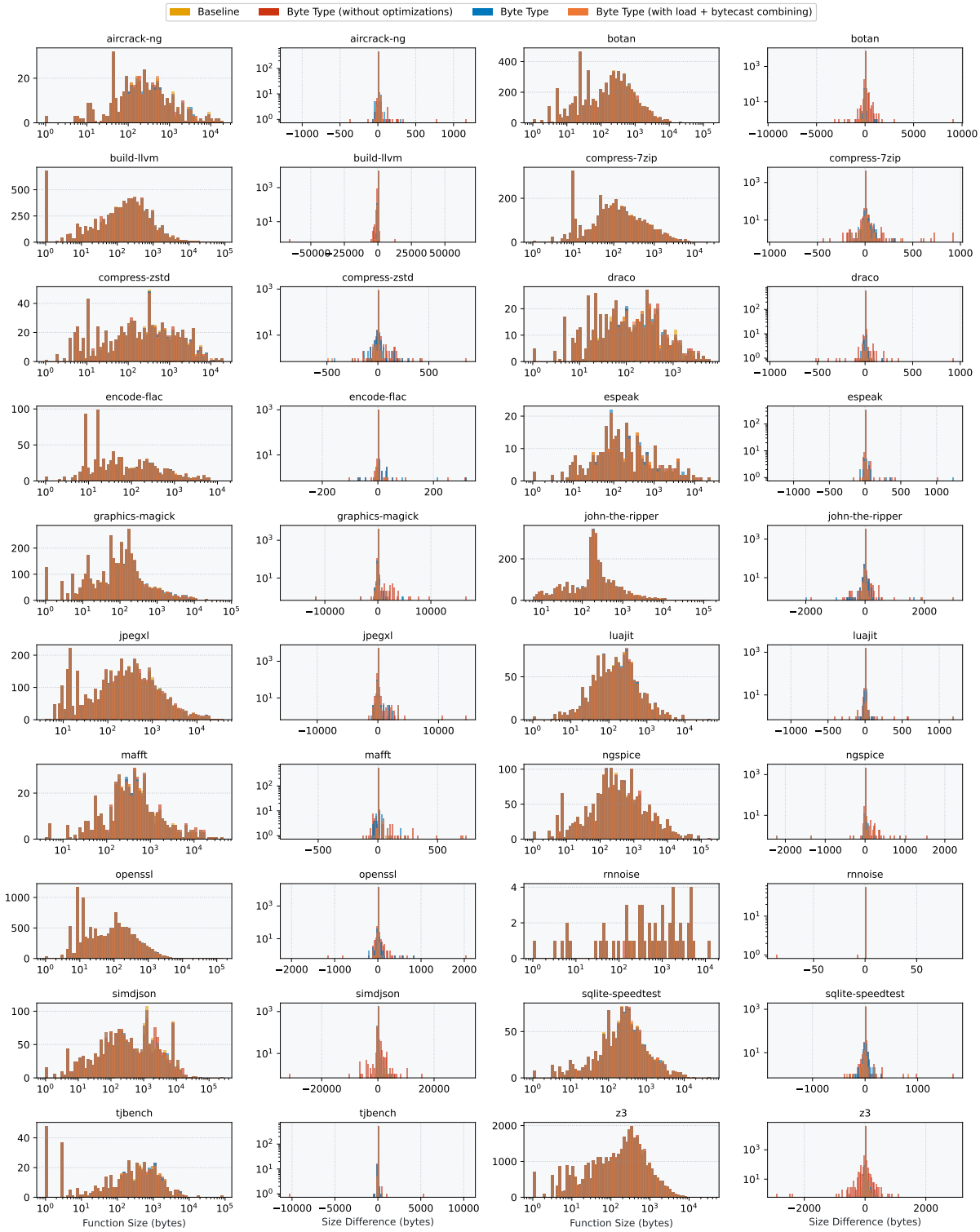


Figure 8.2: Histograms (1st and 3rd columns) showing the distribution of function sizes for each benchmark. The 2nd and 4th columns show the histogram of function size differences between the baseline and each prototype variant. A bar at zero is ideal, indicating no change in function size. Bars use translucent colors, so overlapping values blend (e.g., dark orange results from overlapping baseline and other variants).

Root Cause	Benchmark	Impact
Loop vectorization	compress-pbzip2	2.65%
	graphics-magick	1.54%
	tjbench	1.12%
	compress-7zip	$< 1\%$
	espeak	$< 1\%$
Jump threading	simdjson	4.86%
Bitwise byte manipulation	graphics-magick	$< 1\%$
	openssl	$< 1\%$
Loop idiom recognize	espeak	$< 1\%$

Table 8.2: Root causes for the most significant binary size variations observed in benchmarks.

Throughout the remainder of this section, we explore the most significant observed variations in binary size, identifying their root cause and how these were addressed. Table 8.2 summarizes the results. Out of the presented 9 cases, 5 were caused by over-vectorization, which were fixed by tweaking the vectorization cost model. The remaining cases were caused by optimizations blocked by the byte type, which prevented load combinations, promotions of allocations to SSA registers, and inhibited common loop idioms, such as iteration through a null-terminated C string, from being recognized and optimized. The latter optimizations generally contribute to a decrease in code size.

8.4.1 Loop Vectorization

Loop vectorization converts scalar instructions within loops into SIMD instructions, enabling multiple operations to be executed in parallel. This optimization was the main contributor to the increase in binary size observed across the evaluated benchmarks, namely in `compress-pbzip2` (2.7%), `graphics-magick` (1.5%), and `tjbench` (1.1%). In these programs, our implementation considered some vectorization opportunities as profitable, while the baseline version of LLVM did not, due to different costs being assigned to byte and integer loads. Our analysis focuses on a representative example, as these regressions share a common underlying cause.

In the `compress-pbzip2` benchmark, over-vectorization increased the size of a function by 15%. A simplified version of the vectorized loop is shown in Listing 66. It iterates over a 16x16 matrix, checking each row to determine whether any of its elements is non-zero. When a non-zero element is found, the `inUse16` array is updated to indicate the presence of non-zero values within that row.

LLVM fully vectorizes the inner loop, as illustrated in Listing 67. This vectorization requires fully unrolling both the inner and outer loops. For each matrix column, 16 scalar loads are emitted, one for each row index at the same column offset. These scalar values are assembled into a 16-element vector using

```
unsigned char inUse16[16];
for (unsigned i = 0; i < 16; i++) {
  inUse16[i] = 0;
  for (unsigned j = 0; j < 16; j++)
    if (s->inUse[i * 16 + j]) inUse16[i] = 1;
}
```

Listing 66: Loop performing a row-wise reduction over a 16x16 matrix, in the `pbzip2` program

a sequence of `insertelement` operations, starting on line 11. The resulting vector is then compared with zero, on line 17, to identify non-zero elements. The final result is computed by applying a series of `select` instructions to reduce the per-column comparisons into a single boolean vector. This vector, indicating whether any non-zero element was encountered in the row, is stored to the corresponding entry of the `inUse16` vector.

```
1  %s.inUse.1.0  = getelementptr i8, ptr %s.inUse, i64 16
2  %s.inUse.2.0  = getelementptr i8, ptr %s.inUse, i64 32
3  ...
4  %s.inUse.15.0 = getelementptr i8, ptr %s.inUse, i64 240
5  %byte.0.0     = load b8, ptr %s.inUse
6  %byte.1.0     = load b8, ptr %s.inUse.1.0
7  %byte.2.0     = load b8, ptr %s.inUse.2.0
8  ...
9  %byte.15.0    = load b8, ptr %s.inUse.15.0
10
11 %vec.0.0      = insertelement <16 x b8> poison, b8 %byte.0.0, i64 0
12 %vec.1.0      = insertelement <16 x b8> %vec.0.0, b8 %byte.1.0, i64 1
13 %vec.2.0      = insertelement <16 x b8> %vec.1.0, b8 %byte.2.0, i64 1
14 ...
15 %vec.15.0     = insertelement <16 x b8> %vec.14.0, b8 %byte.15.0, i64 15
16 %cast.0       = bytecast <16 x b8> %vec.15.0 to <16 x i8>
17 %cmp.0        = icmp ne <16 x i8> %cast.0, zeroinitializer
18
19 ; repeat for j = 1 up to j = 15
20
21 %or.15        = select <16 x i1> %cmp.15, <16 x i1> splat (i1 true), <16 x i1> %cmp.14
22 ...
23 %or.1         = select <16 x i1> %or.2, <16 x i1> splat (i1 true), <16 x i1> %cmp.1
24 %or.0         = select <16 x i1> %or.1, <16 x i1> splat (i1 true), <16 x i1> %cmp.0
25 %or.zext      = zext <16 x i1> %or.0 to <16 x i8>
26 store <16 x i8> %or.zext, ptr %inUse16.i
```

Listing 67: Simplified LLVM IR generated for the vectorized version of a loop in the `compress-pbzip2` benchmark

The baseline version of LLVM does not vectorize this loop. A later optimization unrolls the inner loop, but preserves the outer loop, explaining the observed difference in code size. This behavior is justified by the cost model assigning a higher cost (5) to integer loads in baseline LLVM, compared to the cost assigned to the corresponding byte loads (3) in our implementation. This cost difference accumulates when analyzing the scalar loop, causing the cost model to cross the threshold at which vectorization is considered profitable. By adjusting the cost model to assign equal costs to integer and byte loads, we fixed the observed regressions.

8.4.2 Jump Threading

In the `simdjson` benchmark, a binary size regression of 4.86% was fixed by extending LLVM's jump threading optimization to work over byte constants. The jump threading pass simplifies control flow and removes unnecessary branches by predicting what branches will be taken for a particular control flow. It threads jumps directly to the correct successor, removing redundant conditional checks.

One such form of optimization is achieved by tracking whether a particular value is known to be constant across a given control flow path. In Listing 68, LLVM successfully identifies the two possible control flow paths in the presented function. If, on the left, the condition represented by `%cond` evaluates to `true`, execution proceeds to the `%then` block, where `%ret1` is assigned the value 16. This, in turn, causes the comparison in the `%merge` block to evaluate to `false`, and the function to return the same value. Conversely, if `%cond` is `false`, control transfers directly from the `%entry` block to the `%merge` block, where `%ret2` is assigned the value 0. In this case, the comparison evaluates to `true`, leading the function to return the same value. The function can be optimized, as shown on the right, to directly thread the jump from `%entry` to `%exit.false`, if `%cond` is `false`, or to `%exit.true`, otherwise.

```
define i1 @src(i1 %cond) {
entry:
  br i1 %cond, label %then, label %merge

then:
  %ret1 = phi b32 [ 16, %entry ]
  br label %merge

merge:
  %ret2 = phi b32 [ %ret1, %then ], [ 0, %entry ]
  %i = bytecast b32 %ret2 to i32
  %cmp = icmp eq i32 %i, 0
  br i1 %cmp, label %exit.true, label %exit.false

exit.true:
  ret i1 true

exit.false:
  ret i1 false
}
```

```
define i1 @tgt(i1 %cond) {
entry:
  br i1 %cond, label %exit.false,
               label %exit.true

exit.true:
  ret i1 true

exit.false:
  %ret1 = phi b32 [ 16, %entry ]
  %i = bytecast b32 %ret2 to i32
  ret i1 false
```

Listing 68: Jump threading optimization over byte constants

Previously, this optimization only considered integer constants in predecessor blocks when performing this form of simplification. Extending it to operate on byte constants enabled additional control-flow simplifications of the kind illustrated above. These, in turn, allowed for the promotion of an additional `alloca` instruction, which previously had been inhibited by a `getelementptr` operation that caused the pointer to escape. By enabling the elimination of the `getelementptr` instruction, the `alloca` became eligible for promotion by SROA, reducing the binary size of the function by half.

8.4.3 Bitwise Byte Manipulation

In the `graphics-magick` benchmark, an increase in code size was caused by a store forwarding optimization that was inhibited by the byte type. Individual bytes were loaded from memory and stored into a temporary buffer. This buffer was then byte-swapped and reinterpreted as a `float`. LLVM failed to recognize this pattern and did not eliminate the final reinterpreting load.

The baseline version of LLVM is able to recognize this pattern and optimize it, as the loads are performed through integer types. The compiler eliminates the reinterpreting load, reconstructing its value by combining the loaded integers through shifts and bitwise operations. This combined value is then reinterpreted as a `float` through a `bitcast` instruction. This avoids stores to a temporary buffer and the final reinterpreting load. A simplified version of the IR emitted by baseline LLVM (left) and our implementation (right) is shown in Listing 69.

```
%b.0       = load i8, ptr %p
%buf.3.ext = zext i8 %b.0 to i32
%buf.3.sh  = shl i32 %buf.3.ext, 24
%p.1       = getelementptr i8, ptr %p, i64 1
%b.1       = load i8, ptr %p.1
%buf.2.ext = zext i8 %b.1 to i32
%buf.2.sh  = shl i32 %buf.2.ext, 16
%buf.2.in  = or disjoint i32 %buf.2.sh, %buf.3.sh
%p.2       = getelementptr i8, ptr %p, i64 2
%b.2       = load i8, ptr %p.2
%buf.1.ext = zext i8 %b.2 to i32
%buf.1.sh  = shl i32 %buf.1.ext, 8
%buf.1.in  = or disjoint i32 %buf.2.in, %buf.1.sh
%p.3       = getelementptr i8, ptr %p, i64 3
%b.3       = load i8, ptr %p.3
%buf.0.ext = zext i8 %b.3 to i32
%buf.0.in  = or disjoint i32 %buf.1.in, %buf.0.ext
%res       = bitcast i32 %buf.0.in to float
```

```
%b.0    = load b8, ptr %p
%buf.3  = getelementptr i8, ptr %buf, i64 3
store b8 %b.0, ptr %buf.3
%p.1    = getelementptr i8, ptr %p, i64 1
%b.1    = load b8, ptr %p.1
%buf.2 = getelementptr i8, ptr %buf, i64 2
store b8 %b.1, ptr %buf.2
%p.2    = getelementptr i8, ptr %p, i64 2
%b.2    = load b8, ptr %p.2
%buf.1 = getelementptr i8, ptr %buf, i64 1
store b8 %b.2, ptr %buf.1
%p.3    = getelementptr i8, ptr %p, i64 3
%b.3    = load b8, ptr %p.3
store b8 %b.3, ptr %buf
%res    = load float, ptr %buf
```

Listing 69: IR emitted by baseline LLVM (left) and by our implementation (right) of a function of `graphics-magick`

In the `openssl` benchmark, a binary size regression was caused by a similar issue. A function loads four bytes from memory and reinterprets them as a 64-bit integer value. The IR previously generated by our implementation loaded the 8-bit values as bytes and casted them to integers through several `bytecast` instructions. Then, shifts and bitwise operations were emitted to combine the loads. However, the transformation pass responsible for actually combining the loads was blocked by the `bytecast` instructions. The baseline version of LLVM combines the four loads into a single 32-bit load. Listing 70 shows the code for this function, and the IR previously emitted by baseline LLVM (left) and by our implementation (right).

Both regressions were caused by the inability to perform bitwise operations on byte values. However, as such optimizations simply rearrange raw bytes values and reinterpret them as integral values, `bytecast` instructions were introduced, coercing the raw bytes to integer values. This enabled identical

```
static uint64_t load_4(const uint8_t *in) {
  uint64_t result;
  result  = ((uint64_t)in[0]);
  result |= ((uint64_t)in[1]) << 8;
  result |= ((uint64_t)in[2]) << 16;
  result |= ((uint64_t)in[3]) << 24;
  return result;
}
```

```
%l = load i32, ptr %in
%e = zext i32 %0 to i64
ret i64 %e
```

```
%b.0   = load b8, ptr %in
%i.0   = bytecast b8 %b.0 to i8
%e.0   = zext i8 %i.0 to i64
%in.1  = getelementptr i8, ptr %in, i64 1
%b.1   = load b8, ptr %in.1
%i.1   = bytecast b8 %b.1 to i8
%e.1   = zext i8 %i.1 to i64
%shl.1 = shl i64 %e.1, 8
%or.1  = or disjoint i64 %shl.1, %e.0
%in.2  = getelementptr i8, ptr %in, i64 2
%b.2   = load b8, ptr %in.2
%i.2   = bytecast b8 %b.2 to i8
%e.2   = zext i8 %i.2 to i64
%shl.2 = shl nuw nsw i64 %e.1, 16
%or.2  = or disjoint i64 %or.1, %shl.2
%in.3  = getelementptr i8, ptr %in, i64 3
%b.3   = load b8, ptr %in.3
%i.3   = bytecast b8 %b.3 to i8
%e.3   = zext i8 %i.3 to i64
%shl.3 = shl i64 %e.3, 24
%res   = or disjoint i64 %or.2, %shl.3
ret i64 %res
```

Listing 70: Function in `openssl` (top left) and respective IR (baseline, bottom left; our implementation, right)

optimization over the integer values, addressing the first binary size regression. The second regression was addressed by extending the optimization pass to look through the casts and combine the loads.

8.4.4 Loop Idiom Recognize

In the `espeak` benchmark, a loop was not eliminated, as our implementation failed to identify it as a loop iterating over a C string. The baseline version of LLVM recognizes this idiom and optimizes the loop into a call to `strlen`. The loop in question scans a group of dictionary rules for a match against a substring of an input word. If no matching rule is found, the current group is skipped by iterating over a null-terminated string. Identical iteration patterns over other strings in the same function all failed to be optimized. The loop was not recognized as a `strlen` loop idiom as the underlying element type over which the iteration occurs was the byte type, and not an integer type. Extending the optimization to byte-typed values fixed this regression.

8.5 Alive2

8.5.1 LLVM Test Suite

Adding the byte type to Alive2 enabled the automatic verification of both the reworked and newly added optimizations. Assessing both the correctness of the implementation and the broader impact of introducing the byte type to the IR, we ran Alive2 over the LLVM test suite. We identified 11 tests (8% of the

total) that were previously flagged as unsound by Alive2 in baseline LLVM and are now fixed and listed in Table 8.3. These tests were fixed by using the byte type for lowering `memcpy` functions and optimizations that widen or merge loads. No new test failures were reported. Some additional tests containing unsound optimizations were also addressed. However, Alive2 did not flag them as unsound due to the presence of unsupported features, such as multiple address spaces.

Test	Unsoundness Reason
ExpandMemCmp/AArch64/memcmp.ll	`memcmp` to integer load/store pairs
ExpandMemCmp/X86/bcmp.ll	`bcmp` to integer load/store pairs
ExpandMemCmp/X86/memcmp-x32.ll	`memcmp` to integer load/store pairs
ExpandMemCmp/X86/memcmp.ll	`memcmp` to integer load/store pairs
GVN/metadata.ll	Unsound pointer coercions
GVN/pr24397.ll	Unsound pointer coercions
InstCombine/bcmp-1.ll	`bcmp` to integer load/store pairs
InstCombine/memcmp-1.ll	`memcmp` to integer load/store pairs
InstCombine/memcpy-to-load.ll	`memcpy` to integer load/store pairs
PhaseOrdering/swap-promotion.ll	`memcpy` to integer load/store pairs
SROA/alignment.ll	`memcpy` to integer load/store pairs

Table 8.3: LLVM unit tests (under the `llvm/test/Transforms` directory) that were previously flagged as unsound by Alive2. The second column describes the incorrect transformation exposed by each test.

8.5.2 Single File Programs

We applied translation validation to two single-file C programs: `bzip2`[2] and `sqlite3`[3]. Both programs were compiled with the `-O2` optimization level. Validation time, as shown in Figure 8.3, did not change noticeably in either case. In `bzip2`, no differences were detected during validation. In `sqlite3`, two optimizations previously reported as unsound by Alive2 were fixed. These stem from unsound lowerings of `memcpy` calls to integer load/store pairs. No new issues were reported.

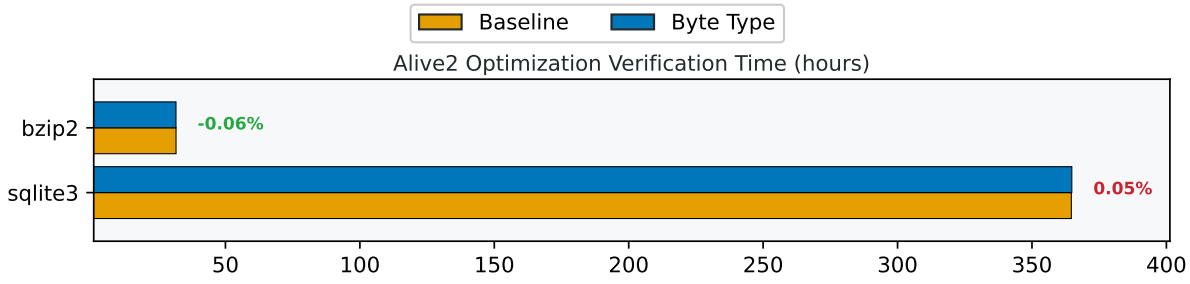


Figure 8.3: Optimization verification times (hours) in Alive2 for the `bzip2` and `sqlite3` programs.

[2]Available at https://people.csail.mit.edu/smcc/projects/single-file-programs/bzip2.c
[3]Available at https://raw.githubusercontent.com/azadkuh/sqlite-amalgamation/refs/heads/master/sqlite3.c

9

Conclusion

Contents

In this thesis, we introduced a dedicated byte type to LLVM IR. This addition addresses a longstanding limitation in LLVM, which lacked an appropriate type for representing and manipulating raw memory values. The new type addresses known miscompilations that plague LLVM, where unnoticed pointer escapes break optimizations due to the implicit conversions that occur upon loading pointer values from memory. By lowering Clang's raw memory access types to the byte type, we preserve their raw memory semantics as specified by the C and C++ standards, and fix such miscompilations.

Furthermore, the byte type also enables the native implementation of memory-related intrinsics in the IR, including `memcpy`, `memmove`, and `memcmp`. We used the new type to fix existing optimizations and implement new ones. By allowing LLVM to represent raw memory values in registers, we provide a solid foundation for future compiler optimizations.

The addition of the byte type also benefits other programming language implementations that leverage LLVM's code generation capabilities. Such languages can benefit not only from the improved correctness granted to the LLVM optimizer, but also from the ability to lower suitable raw memory access types to the byte type, similarly to Clang.

In conclusion, the addition of the byte type to the IR fixes existing transformations and unlocks new optimization opportunities, without incurring any significant performance regressions. Our implementation spans over 2.6k lines of C++ code, corresponding to 0.05% of the codebase of LLVM and Clang. The addition of the new type addresses one of the longstanding limitations of LLVM IR and constitutes a significant step towards a more correct compiler.

9.1 Future Work

The introduction of the byte type to LLVM IR laid the groundwork for more correct optimizations manipulating raw memory. While this thesis shows the followed approach is both practical and beneficial, there remain several opportunities for further exploration and improvement.

The lowering of Clang's raw memory access types to the byte type caused substantial changes in the generated unoptimized IR, due to the widespread use of types such as `char` and `unsigned char` in C and C++ code. These lowerings required the insertion of explicit `bytecast` instructions, reinterpreting byte values as integers in contexts where integer values were previously expected. Although the executed benchmarks and Clang's regression tests cover a wide range of code generation patterns, such coverage will always be incomplete. Fuzz testing techniques could address this limitation by generating C and C++ programs making extensive use of types lowered to the byte type. A dedicated fuzzer could help identify compiler crashes or miscompilations, respectively introduced by missing `bytecast` instructions or incorrect handling of byte values, thereby improving the robustness of the presented solution.

While this work focused primarily on the Clang frontend, a sub-project of LLVM, other projects that leverage LLVM for code generation could also benefit from lowering their raw memory access type to the byte type. The Rust programming language provides the `u8` type, analogous to `unsigned char` in C. Although its raw memory access capabilities are not formally specified, other constructs such as `MaybeUninit<T>`[1] explicitly represent uninitialized memory, providing a safer and more defined abstraction over a raw byte, which conceptually aligns with the byte type. Extending the byte type support to `rustc` and other frontends would enhance consistency across the LLVM ecosystem, potentially fixing existing bugs and unlocking new optimization opportunities in those languages as well.

All performance benchmarks in this thesis were conducted on an x86-64 system. While the results indicate that the introduction of the byte type does not degrade performance on this particular architecture, under the benchmarked applications, LLVM targets a wide range of other widely used architectures, such as AArch64. The impact of the byte type on these architectures remains unexamined and could differ due to target-specific middle-end or backend optimizations. Comprehensive benchmarking across these architectures would benefit the proposed solution.

[1] https://doc.rust-lang.org/beta/std/mem/union.MaybeUninit.html

The introduction of the byte type required extensive changes across various parts of the compiler, throughout the frontend and optimizer. A myriad of tests within the Clang test suite were affected. Due to the sheer amount of affected tests and given that LLVM is a continuously evolving project, maintaining and updating these modifications demands a constant effort to keep pace with upstream developments. The massive number of changes requires a thorough review by a diverse group of maintainers, before ultimately upstreaming the byte type to LLVM.

The byte type proposal could be simplified by adopting an alternative semantics to `load` instructions. By returning a pointer with no provenance upon loading a pointer through an integer type, the `exact` version of the `bytecast` would no longer be needed. Storing a byte value to memory and loading it back through another type would be equivalent to performing a regular `bytecast` to that same type. This alternative semantics would enable `load`/`bytecast` pairs to be combined, as depicted in Listing 71. Furthermore, the current lowering of `memcmp` to integer loads would soundly compare pointers values, as these would be implicitly reinterpreted as integers. Overall, this alternative semantics is worth contemplating as it avoids the need to define two different variants of the `bytecast`. Furthermore, no alias analysis precision is lost, as the loaded pointer has no provenance.

```
%b = load b8, ptr %p
%i = bytecast b8 %b to i8
```

```
%i = load i8, ptr %p
```

Listing 71: Combining of `load` and `bytecast` pairs

Bibliography

[1] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, "Taming undefined behavior in LLVM," ser. PLDI, 2017. [Online]. Available: https://doi.org/10.1145/3062341.3062343

[2] J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis, "A formal C memory model supporting integer-pointer casts," ser. PLDI, 2015. [Online]. Available: https://doi.org/10.1145/2737924.2738005

[3] S. Chakraborty and V. Vafeiadis, "Formalizing the concurrency semantics of an llvm fragment," in *CGO*, 2017. [Online]. Available: https://doi.org/10.1109/CGO.2017.7863732

[4] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the llvm intermediate representation for verified program transformations," in *POPL*, 2012. [Online]. Available: https://doi.org/10.1145/2103656.2103709

[5] J. Lee, C.-K. Hur, R. Jung, Z. Liu, J. Regehr, and N. P. Lopes, "Reconciling high-level optimizations and low-level code in LLVM," *Proc. ACM Program. Lang.*, no. OOPSLA, 2018. [Online]. Available: https://doi.org/10.1145/3276495

[6] C. Beck, I. Yoon, H. Chen, Y. Zakowski, and S. Zdancewic, "A Two-Phase Infinite/Finite Low-Level Memory Model: Reconciling Integer-Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction," *Proc. ACM Program. Lang.*, 2024. [Online]. Available: https://doi.org/10.1145/3674652

[7] Y. Zakowski, C. Beck, I. Yoon, I. Zaichuk, V. Zaliva, and S. Zdancewic, "Modular, compositional, and executable formal semantics for llvm ir," *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Aug. 2021. [Online]. Available: https://doi.org/10.1145/3473572

[8] L. Li and E. L. Gunter, "K-LLVM: A Relatively Complete Semantics of LLVM IR," in *ECOOP*, 2020. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2020.7

[9] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, “Provably correct peephole pptimizations with alive,” in *PLDI*, 2015. [Online]. Available: https://doi.org/10.1145/2737924.2737965

[10] J. Taneja, Z. Liu, and J. Regehr, “Testing static analyses for precision and soundness,” in *CGO*, 2020. [Online]. Available: https://doi.org/10.1145/3368826.3377927

[11] J. Lee, D. Kim, C.-K. Hur, and N. P. Lopes, “An smt encoding of llvm’s memory model for bounded translation validation,” in *CAV*, 2021. [Online]. Available: https://doi.org/10.1007/978-3-030-81688-9_35

[12] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in C compilers,” 2011. [Online]. Available: https://doi.org/10.1145/1993316.1993532

[13] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, “Test-case reduction for C compiler bugs,” ser. PLDI, 2012. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[14] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, “Alive2: bounded translation validation for LLVM,” ser. PLDI, 2021. [Online]. Available: https://doi.org/10.1145/3437992.3439903

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[16] F. Rastello and F. B. Tichadou, *SSA-based Compiler Design*. Springer, 2022. [Online]. Available: https://doi.org/10.1007/978-3-030-80515-9

[17] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau, “Simple and efficient construction of static single assignment form,” in *CC*, 2013. [Online]. Available: https://doi.org/10.1007/978-3-642-37051-9_6

[18] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell, “Exploring C semantics and pointer provenance,” *Proc. ACM Program. Lang.*, no. POPL, 2019. [Online]. Available: https://doi.org/10.1145/3290380

[19] P. Sewell, K. Memarian, and V. B. F. Gomes, “C provenance semantics: detailed semantics (for PNVI-plain, PNVI address-exposed, PNVI address-exposed user-disambiguation, and PVI models),” ISO/IEC JTC1/SC22/WG14, Tech. Rep., 2019. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2364.pdf

[20] Gustedt, Jens and Sewell, Peter and Memarian, Kayvan and Gomes, Victor BF and Uecker, Martin, “A Provenance-aware Memory Object Model for C,” Technical report, Tech. Rep., 2022. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3005.pdf

[21] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, “Stacked borrows: an aliasing model for Rust,” *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 2019. [Online]. Available: https://doi.org/10.1145/3371109

[22] V. Zaliva, K. Memarian, B. Campbell, R. Almeida, N. Filardo, I. Stark, and P. Sewell, “A cheri c memory model for verified temporal safety,” in *CPP*, 2025. [Online]. Available: https://doi.org/10.1145/3703595.3705878

[23] R. Lepigre, M. Sammler, K. Memarian, R. Krebbers, D. Dreyer, and P. Sewell, “Vip: verifying real-world c idioms with integer-pointer casts,” *Proc. ACM Program. Lang.*, vol. 6, no. POPL, Jan. 2022. [Online]. Available: https://doi.org/10.1145/3498681

[24] Y. Kim, M. Cho, J. Lee, J. Kim, T. Yoon, Y. Song, and C.-K. Hur, “Archmage and compcertcast: End-to-end verification supporting integer-pointer casting,” *Proc. ACM Program. Lang.*, vol. 9, no. POPL, Jan. 2025. [Online]. Available: https://doi.org/10.1145/3704881

[25] X. Leroy and S. Blazy, “Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations,” *Journal of Automated Reasoning*, 2008. [Online]. Available: https://doi.org/10.1007/s10817-008-9099-0

[26] X. Leroy, “Formal verification of a realistic compiler,” *Commun. ACM*, 2009. [Online]. Available: https://doi.org/10.1145/1538788.1538814

[27] F. Besson, S. Blazy, and P. Wilke, “CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics,” *Journal of Automated Reasoning*, 2019. [Online]. Available: https://doi.org/10.1007/s10817-018-9496-y

[28] ——, “A Verified CompCert Front-End for a Memory Model Supporting Pointer Arithmetic and Uninitialised Data,” *Journal of Automated Reasoning*, 2019. [Online]. Available: https://doi.org/10.1007/s10817-017-9439-z