Taming Undefined Behavior in LLVM

Juneyoung Lee
Yoonseung Kim
Youngju Song
Chung-Kil Hur
Seoul National University, Korea
{juneyoung.lee, yoonseung.kim, youngju.song, gil.hur}@sf.snu.ac.kr

Sanjoy Das
Azul Systems, USA
sanjoy@azul.com

David Majnemer
Google, USA
majnemer@google.com

John Regehr
University of Utah, USA
regehr@cs.utah.edu

Nuno P. Lopes
Microsoft Research, UK
nlopes@microsoft.com

Abstract

A central concern for an optimizing compiler is the design of its intermediate representation (IR) for code. The IR should make it easy to perform transformations, and should also afford efficient and precise static analysis.

In this paper we study an aspect of IR design that has received little attention: the role of undefined behavior. The IR for every optimizing compiler we have looked at, including GCC, LLVM, Intel's, and Microsoft's, supports one or more forms of undefined behavior (UB), not only to reflect the semantics of UB-heavy programming languages such as C and C++, but also to model inherently unsafe low-level operations such as memory stores and to avoid over-constraining IR semantics to the point that desirable transformations become illegal. The current semantics of LLVM's IR fails to justify some cases of loop unswitching, global value numbering, and other important "textbook" optimizations, causing long-standing bugs.

We present solutions to the problems we have identified in LLVM's IR and show that most optimizations currently in LLVM remain sound, and that some desirable new transformations become permissible. Our solutions do not degrade compile time or performance of generated code.

CCS Concepts • **Theory of computation** → **Semantics and reasoning**; • **Software and its engineering** → **Compilers**; **Semantics**

Keywords compilers, undefined behavior, intermediate representations

DOI: http://dx.doi.org/10.1145/3062341.3062343

1. Introduction

Some programming languages, intermediate representations, and hardware platforms define a set of erroneous operations that are untrapped and that may cause the system to behave badly. These operations, called *undefined behaviors*, are the result of design choices that can simplify the implementation of a platform, whether it is implemented in hardware or software. The burden of avoiding these behaviors is then placed upon the platform's users. Because undefined behaviors are untrapped, they are insidious: the unpredictable behavior that they trigger often only shows itself much later.

The AVR32 processor architecture document [2, p. 51] provides an example of hardware-level undefined behavior:

> If the region has a size of 8 KB, the 13 lowest bits in the start address must be 0. Failing to do so will result in UNDEFINED behaviour.

The hardware developers have no obligation to detect or handle this condition. ARM and x86 processors (and, indeed, most CPUs that we know of) also have undefined behaviors.

At the programming language level, Scheme R6RS [24, p. 54] mentions that "The effect of passing an inappropriate number of values to such a continuation is undefined." However, the best-known examples of undefined behaviors in programming languages come from C and C++, which have hundreds of them, ranging from simple local operations (overflowing signed integer arithmetic) to global program behaviors (race conditions and violations of type-based aliasing rules). Undefined behaviors facilitate optimizations by permitting a compiler to assume that programs will only execute defined operations, and they also support error checking, since a conforming compiler implementation can cause the program to abort with a diagnostic when an undefined operation is executed.

Some intermediate representations (IRs), such as Java bytecode, have been designed to minimize or eliminate undefined behaviors. On the other hand, compilers often support one or more forms of undefined behavior in their

IR: all of LLVM, GCC, the Intel C/C++ compiler, and the Microsoft Visual C++ compiler do. At this level, undefined behavior must walk a very thin line: the semantics of the IR have to be tight enough that source-level programs can be efficiently compiled to IR, while also being weak enough to allow desirable IR-level optimizations to be performed and to facilitate efficient mapping onto target instruction sets.

Undefined behavior (UB) in LLVM falls into two categories. First, "immediate UB" for serious errors, such as dividing by zero or dereferencing an invalid pointer, that have consequences such as a processor trap or RAM corruption. Second, "deferred UB" for operations that produce unpredictable values but are otherwise safe to execute. Deferred UB is necessary to support speculative execution, such as hoisting potentially undefined operations out of loops. Deferred UB in LLVM comes in two forms: an *undef* value that models a register with indeterminate value, and *poison*, a slightly more powerful form of UB that taints the dataflow graph and triggers immediate UB if it reaches a side-effecting operation.

The presence of two kinds of deferred UB, and in particular the interaction between them, has often been considered to be unsatisfying, and has been a persistent source of discussions and bugs. LLVM has long contained optimizations that are inconsistent with the documented semantics and that are inconsistent with each other. The LLVM community has dealt with these issues by usually fixing a problem when it can be demonstrated to lead to an *end-to-end miscompilation*: an optimizer malfunction that leads to the wrong machine code being emitted for some legal source program. The underlying, long-standing problems—and, indeed, a few opportunities for end-to-end miscompilations—remain unsolved.

To prevent miscompilation, to permit rigorous reasoning about LLVM IR, and to support formal-methods-based tools such as superoptimizers, translation validators, and program verifiers, we have attempted to redefine the UB-related parts of LLVM's semantics in such a way that:

- Compiler developers can understand and work with the semantics.
- Long-standing optimization bugs can be fixed.
- Few optimizations currently in LLVM need to be removed.
- Compilation time and execution time of generated code are largely unaffected.
- The LLVM code base can be migrated to the new semantics in a series of modest stages that cause very little breakage.

This paper describes and evaluates our ongoing efforts.

2. Undefined Behavior in the IR

Undefined-behavior-related compiler optimizations are often thought of as black magic, even by compiler developers. In this section we introduce IR-level undefined behavior and show examples where it enables useful optimizations.

2.1 Undefined Behavior $\neq$ Unsafe Programming

Despite the very poor example set by C and C++, there is no inherent connection between undefined behavior (UB) and unsafe programming. Rather, UB simply reflects a refusal to systematically trap program errors at one particular level of the system: the responsibility for avoiding these errors is delegated to a higher level of abstraction. For example, of course, many safe programming languages have been compiled to machine code, the unsafety of which in no way compromises the high-level guarantees made by the language implementation. Swift and Rust are compiled to LLVM IR; some of their safety guarantees are enforced by dynamic checks in the emitted code, other guarantees are made through type checking and have no representation at the LLVM level. Even C can be used safely if some tool in the development environment ensures—either statically or dynamically—that it will not execute UB.

The essence of undefined behavior is the freedom to avoid a forced coupling between error checks and unsafe operations. The checks, once decoupled, can be optimized, for example by being hoisted out of loops or eliminated outright. The remaining unsafe operations can be—in a well-designed IR—mapped onto basic processor operations with little or no overhead. As a concrete example, consider this Swift code:

```
func add(a : Int, b : Int)->Int {
  return (a & 0xffff) + (b & 0xffff)
}
```

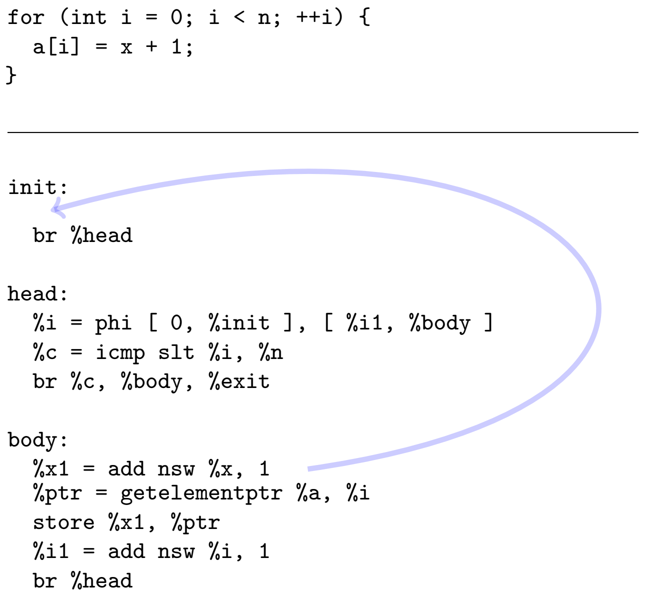
Although a Swift implementation must trap on integer overflow, the compiler observes that overflow is impossible and emits this LLVM IR:

```
define i64 @add(i64 %a, i64 %b) {
  %0 = and i64 %a, 65535
  %1 = and i64 %b, 65535
  %2 = add nuw nsw i64 %0, %1
  ret i64 %2
}
```

Not only has the checked addition operation been lowered to an unchecked one, but in addition the `add` instruction has been marked with LLVM's `nsw` and `nuw` attributes, indicating that both signed and unsigned overflow are undefined. In isolation these attributes provide no benefit, but they may enable additional optimizations after this function is inlined. When the Swift benchmark suite[1] is compiled to LLVM, about one in eight addition instructions has an attribute indicating that integer overflow is undefined.

In this particular example the `nsw` and `nuw` attributes are redundant since an optimization pass could re-derive the fact that the `add` cannot overflow. However, in general these attributes and others like them add real value by avoiding the

[1] `https://swift.org/blog/swift-benchmark-suite/`



```
for (int i = 0; i < n; ++i) {
  a[i] = x + 1;
}
```

```
init:

  br %head

head:
  %i = phi [ 0, %init ], [ %i1, %body ]
  %c = icmp slt %i, %n
  br %c, %body, %exit

body:
  %x1 = add nsw %x, 1
  %ptr = getelementptr %a, %i
  store %x1, %ptr
  %i1 = add nsw %i, 1
  br %head
```

Figure 1. C code and its corresponding LLVM IR. We want to hoist the invariant addition out of the loop. The `nsw` attribute means the `add` is undefined for signed overflow.

need for potentially expensive static analyses to rediscover known program facts. Also, some facts cannot be rediscovered later, even in principle, since information is lost at some compilation steps.

2.2 Enabling Speculative Execution

The C code in Figure 1 executes undefined behavior if `x` is `INT_MAX` and $n > 0$, because in this case the signed addition `x + 1` overflows. A straightforward translation of the C code into LLVM IR, also shown in Figure 1, has the same domain of definedness as the original code: the `nsw` modifier to the `add` instruction indicates that it is defined only when signed overflow does not occur.

We would like to optimize the loop by hoisting the invariant expression `x + 1`. If integer overflow triggered immediate undefined behavior, this transformation would be illegal because it makes the domain of definedness smaller: the code would execute UB when `x` was `INT_MAX`, even if `n` was zero. LLVM works around this problem by adding the concept of deferred undefined behavior: the undefined addition is allowed, but the resulting value cannot be relied upon. It is easy to see that after hoisting the `add`, the code remains safe in the $n = 0$ case, because `x1` is not used. While deferred UB is useful, it is not appropriate in all situations. For example, division by zero can trigger a processor trap and an out-of-bounds store can corrupt RAM. These operations, and a few others in LLVM IR, are immediate undefined behaviors and programs must not execute them.

2.3 Undefined Value

We need a semantics for deferred undefined behavior. A reasonable choice is to specify that an undefined value represents any value of the given type. A number of compiler IRs support this abstraction; in LLVM it is called undef.[2]

Undef is useful because it lets the compiler avoid materializing an arbitrary constant in situations—such as the one shown in Figure 2—where the exact value does not matter. In this example, assume that `cond2` implies `cond` in some non-trivial way such that the compiler cannot see it. Thus, there is no need to initialize variable `x` at its point of declaration since it is only passed to `g` after being assigned a value from `f`'s return value. If the IR lacked an undef value, the compiler would have to use an arbitrary constant, perhaps zero, to initialize `x` on the branch that skips the first `if` statement. This, however, increases the code size by one instruction and two bytes on x86 and x86-64. Little optimizations like this can add up across a large program.

LLVM uses undef in a few other situations. First, to represent the values of padding in structures, arrays, and bit fields. Second, operations such as shifting a 32-bit value by 33 places evaluate to undef. LLVM's shift operators result in deferred UB for shift-past-bitwidth because different processors produce different results for this condition; mandating any particular behavior would require some platforms to introduce potentially expensive code.

2.4 Beyond Undef

In C and C++, we can assume that the expressions `a + b > a` and `b > 0` always yield the same value because signed overflow is undefined (assuming `a` and `b` are of a signed type like `int`). If the original expression is translated to this LLVM IR:

```
%add = add %a, %b
%cmp = icmp sgt %add, %a
```

the optimization to:

```
%cmp = icmp sgt %b, 0
```

becomes illegal since the `add` instruction wraps around on overflow. Moreover, this problem cannot be fixed by defining a version of `add` that returns `undef` when there is a signed integer overflow.

To see the inadequacy of undef, let $\texttt{a} = \texttt{INT\_MAX}$ and $\texttt{b} = 1$. The addition overflows and the expression simplifies to $\texttt{undef} > \texttt{INT\_MAX}$, which is always false since there is no value of integer type that is larger than `INT_MAX`. However, the desired optimized expression, $\texttt{b} > 0$, simplifies to $1 > 0$, which is true. Thus, the optimization is illegal: it would change the semantics of the program.

To justify this transformation, LLVM has a second kind of deferred undefined behavior, the *poison value*. The original expression is compiled to this code instead:

[2] `http://nondot.org/sabre/LLVMNotes/UndefinedValue.txt`

```
int x;
if (cond)
  x = f();

if (cond2)
  g(x);
```

(a)

```
entry:
  br %cond, %ctrue, %cont

ctrue:
  %xf = call @f()
  br %cont

cont:
  %x = phi [ %xf, %ctrue ], [ undef, %entry ]
  br %cond2, %c2true, %exit

c2true:
  call @g(%x)
```

(b)

```
  ; test cond
  testb   %dil, %dil
  je      ctrue
  ; return value goes in %eax
  callq   f

ctrue:
  ; test cond2
  testb   %bl, %bl
  je      exit
  ; whatever is in %eax
  ; gets passed to g()
  movl    %eax, %edi
  callq   g
```

(c)

Figure 2. If cond2 implies cond, the C code in (a) does not perform UB by accessing x before it is assigned a value. (b) is Clang's translation into LLVM IR and (c) is the eventual x86-64.

```
for (int i = 0; i <= n; ++i) {
  a[i] = 42;
}
```

```
entry:
  br %head

head:
  %i = phi [ 0, %entry ], [ %i1, %body ]
  %c = icmp sle %i, %n
  br %c, %body, %exit

body:
  %iext = sext %i to i64
  %ptr = getelementptr %a, %iext
  store 42, %ptr
  %i1 = add nsw %i, 1
  br %head
```

Figure 3. C code and corresponding LLVM IR on x86-64. We want to eliminate the sext instruction in the loop body.

```
%add = add nsw %a, %b
%cmp = icmp sgt %add, %a
```

The nsw (no signed wrap) attribute on the add instruction indicates that it returns a poison value on signed overflow. Poison values, unlike undef, are not restricted to being a value of a given type. Most instructions including icmp return poison if any of their inputs is poison. Thus, poison is a stronger form of UB than undef. In the previous example with nsw, the result of the comparison becomes poison whenever the addition overflows and thus the optimization is justified.

Figure 3 shows another example motivating the poison value. The getelementptr instruction (GEP for short) performs pointer arithmetic. The GEP there is computing $\mathtt{a}+\mathtt{i}*4$, assuming that a is an array of 4-byte integers.

The sign-extend operation sext in the loop body handles the mismatch in bitwidth between the 32-bit induction variable and the 64-bit pointer size. It is the low-level equivalent of casting an int to long in C. Therefore, the GEP in the program is actually computing $\mathtt{a} + \mathtt{sext}(\mathtt{i}) * 4$. We would like to optimize away the sext instruction since sign extension, unlike zero extension, is usually not free at runtime.

If we convert the loop induction variable i into long we can remove the sign extension within the loop body (at the expense of adding a sign extend of n to the entry basic block). This transformation improves performance by up to 39%, depending on the microarchitecture, since we save one instruction per iteration (cltq — sign extend eax into rax).

The transformation is only valid if pointer arithmetic overflow is undefined. If it is defined to wrap around, the transformation is not semantics-preserving, since a sequence of values of a signed 32-bit counter is different from a signed 64-bit counter's. Therefore, we would be changing the set of stored locations in case of overflow.

For a compiler to perform the aforementioned transformation, it needs to prove that either the induction variable does not overflow, or if it does it is a signed operation and therefore it does not matter. As we have seen before, signed integer overflow cannot be immediate UB since that would prevent hoisting math out of loops. If signed integer overflow returns undef, the resulting semantics are too weak to justify the desired optimization: on overflow we would obtain $\mathtt{sext}(\mathtt{undef})$ for %iext, which has all the most-significant bits equal to either zero or one. Therefore, the maximum value %i1 could take would be INT_MAX and thus the comparison at %c would always be true if $\%\mathtt{n} = \mathtt{INT\_MAX}$. On the other hand, the comparison with 64-bit integers would return false instead.

If overflow is defined to return poison, an induction variable overflow would result in $\texttt{\%iext} = \texttt{sext}(\texttt{poison})$, which is equal to `poison`, which would make the comparison at `%c` equal to `poison` as well. Therefore, this semantics justifies induction variable widening.

3. Inconsistencies in LLVM

In this section we present several examples of problems with the current LLVM IR semantics.

3.1 Duplicate SSA uses

In some CPU micro-architectures, addition is cheaper than multiplication. It may therefore be beneficial to rewrite $2 \times x$ as $x + x$. In LLVM IR we want to rewrite:

```
%y = mul %x, 2
```

as:

```
%y = add %x, %x
```

Algebraically, these two expressions are equivalent. However, consider the case where `%x` is undef. In the original code, the result can be any even number, while in the transformed code the result can be any number. Therefore, the transformation is wrong because we have increased the set of possible outcomes.

This problem happens because each use of undef in LLVM can yield a different result. Therefore, it is not correct in general to increase the number of uses of a given SSA register in an expression tree, unless it can be proved to not hold the undef value. Even so, LLVM incorrectly performs similar transformations.

There are, however, multiple advantages to defining `undef` as yielding a possibly different value on each use. For example, it helps reduce register pressure since we do not need to hold the value of an `undef` in a register to give the same value to all uses. Secondly, peephole optimizations can easily assume that an `undef` takes whatever value is convenient to do a particular transformation, which they could not easily do if `undef` had to remain consistent over multiple uses. Another advantage is to allow duplication of memory loads given that loads from uninitialized memory yield `undef`. If `undef` was defined to return a consistent value for all uses, a duplicated load could potentially return a different value if loading from uninitialized memory, which would be incorrect.

3.2 Hoisting operations past control-flow

Consider this example:

```
if (k != 0) {
  while (c) {
    use(1 / k);
  }
}
```

Since $1/\texttt{k}$ is loop invariant, LLVM would like to hoist it out of the loop. Hoisting the division seems safe because the top-level if-statement ensures that division by zero will not happen. This gives:

```
if (k != 0) {
  int t = 1 / k;
  while (c) {
    use(t);
  }
}
```

Now consider the case where `k` is undef. Since each use of undef can yield a different result, we can have the top-level if-condition being true and still divide by zero, when this could not have happened in the original program if the execution never reached the division (e.g., if `c` was false). Thus, this transformation is unsound. LLVM used to do it, but stopped after it was shown to lead to end-to-end miscompilation.[3]

3.3 Global Value Numbering vs. Loop Unswitching

When `c2` is loop-invariant, LLVM's loop unswitching optimization transforms code of this form:

```
while (c) {
  if (c2) { foo }
  else    { bar }
}
```

to:

```
if (c2) {
  while (c) { foo }
} else {
  while (c) { bar }
}
```

This transformation assumes that branching on poison is not UB, but is rather a non-deterministic choice. Otherwise, if `c2` was poison, then loop unswitching would be introducing UB if `c` was always false (i.e., if the loop never executed).

The goal of global value numbering (GVN) is to find equivalent expressions and then pick a representative one and remove the remaining (redundant) computations. For example, in the following code, variables `t`, `w`, and `y` all hold the same value within the "then" block:

```
t = x + 1;
if (t == y) {
  w = x + 1;
  foo(w);
}
```

Therefore, GVN can pick `y` as the representative value and transform the code into:

```
t = x + 1;
if (t == y) {
  foo(y);
}
```

[3] `http://llvm.org/PR21412`

However, if `y` is a poison value and `w` is not, we have changed the code from using a regular value as function argument to passing a poison value to `foo`. If GVN followed loop unswitching's interpretation of branch-on-poison (non-deterministic branch), the transformation would be unsound. However, if we decide instead that branch-on-poison is UB, then GVN is fine, since the comparison "`t == y`" would be poison and therefore the original program would be already executing UB. This, however, contradicts the assumption made by loop unswitching. In other words, loop unswitching and GVN require different semantics for branch on poison in LLVM IR in order to be correct. By assuming different semantics, they perform conflicting optimizations, enabling end-to-end miscompilations.[4]

3.4 Select and Poison

LLVM's ternary `select` instruction, like the `?:` operator in C/C++, uses a Boolean to choose between its arguments. Either choice for how `select` deals with poison—producing poison if its not-selected argument is poison, or not—could be used as the basis for a correct optimizer. However, LLVM's optimization passes have not consistently implemented either choice. The LLVM Language Reference Manual[5] implies that if either argument to a `select` is poison, the output is poison.

The SimplifyCFG pass tries to convert control flow into select instructions:

```
  br %cond, %true, %false
true:
  br %merge
false:
  br %merge
merge:
  %x = phi [ %a, %true ], [ %b, %false ]
```

Gets transformed into:

```
  br %merge
merge:
  %x = select %cond, %a, %b
```

For this transformation to be correct, select on poison cannot be UB if branching on poison is not. Moreover, it can only be poison when the chosen value at runtime is poison (in order to match the behavior of phi).

LLVM also performs the reverse transformation, usually late in the pipeline and for target ISAs where it is preferable to branch rather than do a conditional move. For this transformation to be correct, branch on poison can only be UB if select on a poison condition is also UB. Since we want both transformations to be feasible, we can conclude that the behavior of branching on poison and select with a poison condition has to be equivalent.

If select on a poison condition is UB, it makes it very hard for the compiler to introduce select instructions in replacement of arithmetic. E.g., the following transformation that replaces an unsigned division with a comparison would be invalid (which ought to be valid for any constant $\mathtt{C} < 0$):

```
%r = udiv %a, C
```

to:

```
%c = icmp ult %a, C
%r = select %c, 0, 1
```

This transformation is desirable since it removes a potentially expensive operation like division. However, if select on poison is UB, the transformed program would execute UB if `%a` was poison, while the original program would not. As we have seen previously, if select (and therefore branch) on poison is not UB, GVN is unsound, but that is incompatible with the transformation above.

Finally, it is often desirable to view select as arithmetic, allowing transformations like: `%x = select %c, true, %b` to `%x = or %c, %b`. This property of equivalence with arithmetic, however, requires making the return value poison if any of the arguments is poison, which breaks soundness for the phi/branch to select transformation (SimplifyCFG in LLVM) above.

There is a tension between the different semantics that select can take and which optimizations can be made sound. Currently, different parts of LLVM implement different semantics for select, which originates end-to-end miscompilations. [6]

Finally, it is very easy to make mistakes when both undef and poison are involved. LLVM currently performs the following substitution:

```
%v = select %c, %x, undef
```

to:

```
%v = %x
```

This is wrong because `%x` could be poison, and poison is stronger than undef.[7]

3.5 Summary

In this section we showed that undefined behavior, which was added to LLVM's IR to justify certain desirable transformations, is exceptionally tricky and has lead to conflicting assumptions among compiler developers. These conflicts are reflected in the code base.[8] Although the LLVM developers almost always fix overt problems that can be demonstrated to lead to end-to-end miscompilations, the latent problems we have shown here are long-standing and have so far resisted attempts to fix them (any fix that makes too many existing

[4] `http://llvm.org/PR27506` and `http://llvm.org/PR31652`

[5] `http://llvm.org/docs/LangRef.html`

[6] `http://llvm.org/PR31632`

[7] `http://llvm.org/PR31633`

[8] e.g., `http://llvm.org/PR31181` and `http://llvm.org/PR32176`

optimizations illegal is unacceptable). In the next section we introduce a modified semantics for UB in LLVM that we believe fixes all known problems and is otherwise acceptable.

4. Proposed Semantics

In Section 2 we showed that undef and poison enable useful optimizations that programmers might expect. In Section 3, however, we showed that undef and poison, as currently defined, are inconsistent with other desirable transformations (or combinations of transformations) and that they interact poorly with each other. Our proposal—arrived at after many iterations and much discussion, and currently under discussion with the broader LLVM community—is to tame undefined behavior in LLVM as follows:

- Remove undef and use poison instead.
- Introduce a new instruction:

  ```
  %y = freeze %x
  ```

  `freeze` is a nop unless its input is poison, in which case it non-deterministically chooses an arbitrary value of the type. All uses of a given freeze return the same value, but different freezes of a value may return different constants.
- All operations over poison unconditionally return poison except `phi`, `select`, and `freeze`.
- Branching on poison is immediate UB.

Our experience is that the presence of two kinds of deferred undefined behavior is simply too difficult for developers to reason about: one of them had to go. We define `phi` and `select` to conditionally return poison, and branching on poison to be UB, because these decisions reduce the number of freeze instructions that would otherwise be needed.

Defining branching on poison to be UB further enables analyses to assume that predicates used on branches hold within the target basic block, which would not be possible if we had defined branching on poison to be a non-deterministic choice. For example, for code like `if (x > 0) { /* foo */ }`, we want to allow analyses to assume that `x` is positive within the "then" block (and not positive in the "else" block).

A risk of using freeze is that it disables subsequent optimizations that take advantage of poison. Our observation is that many of these optimizations were illegal anyway, and that it is better to disable them explicitly rather than implicitly. Also, as we show later, we usually do not need to introduce many freeze instructions. We experimentally show that freeze does not unduly impact performance.

4.1 Syntax

Figure 4 gives the partial syntax of LLVM IR statements. LLVM IR is typed, but we omit operand types for brevity (in this section and throughout the paper) when these are implicit or non-essential. The IR includes standard unary/binary arithmetic instructions, load/store operations, a phi node, a comparison operator, multiple type casting instructions, conditional branching, instructions to access and modify vectors, etc. We also include the new **freeze** instruction and the new **poison** value, while removing the old **undef** value.

stmt	::=	*reg* = *inst* | **br** *op*, *label*, *label* | **store** *op*, *op*
inst	::=	*binop* $\overline{attr}$ *op*, *op* | *conv op* | **bitcast** *op* | **select** *op*, *op*, *op* | **icmp** *cond*, *op*, *op* | **phi** *ty*, [*op*, *label*] . . . , [*op*, *label*] | **freeze** *op* | **getelementptr** *op*, . . . , *op* | **load** *op* | **extractelement** *op*, *constant* | **insertelement** *op*, *op*, *constant*
cond	::=	**eq** | **ne** | **ugt** | **uge** | **slt** | **sle**
ty	::=	$\mathbf{i}sz$ | $ty*$ | $<sz \times \mathbf{i}sz>$ | $<sz \times ty*>$
binop	::=	**add** | **udiv** | **sdiv** | **shl** | **and** | **or**
attr	::=	**nsw** | **nuw** | **exact**
op	::=	*reg* | *constant* | **poison**
conv	::=	**zext** | **sext** | **trunc**

Figure 4. Partial syntax of LLVM IR statements. Types include arbitrary bitwidth integers, pointers $ty*$, and vectors $<elems \times ty>$ that have a statically-known number of elements $elems$.

4.2 Semantics

We first define the semantic domains as follows.

$$
\begin{array}{lll}
\mathrm{Num}(sz) & ::= & \{\, i \mid 0 \le i < 2^{sz} \,\} \\
[\![\mathbf{i}sz]\!] & ::= & \mathrm{Num}(sz) \uplus \{\, \mathbf{poison} \,\} \\
[\![ty*]\!] & ::= & \mathrm{Num}(32) \uplus \{\, \mathbf{poison} \,\} \\
[\![\langle sz \times ty \rangle]\!] & ::= & \{0, \ldots, sz-1\} \to [\![ty]\!] \\
\mathrm{Mem} & ::= & \mathrm{Num}(32) \nrightarrow [\![\langle 8 \times \mathbf{i}1 \rangle]\!] \\
\mathrm{Name} & ::= & \{\, \%\mathrm{x}, \%\mathrm{y}, \ldots \} \\
\mathrm{Reg} & ::= & \mathrm{Name} \to \{\, (ty, v) \mid v \in [\![ty]\!] \,\}
\end{array}
$$

Here $[\![ty]\!]$ denotes the set of values of type ty, which are either `poison` or fully defined for base types, and are element-wise defined for vector types. The memory Mem is bitwise defined since it has no associated type. Specifically, Mem partially maps a 32-bit address to a bitwise defined byte (we assume, with no loss of generality, that pointers are 32 bits). The register file Reg maps a name to a type and a value of that type.

We define two meta operations: conversion between values of types and low-level bit representation. These operations are used later for defining semantics of instructions.

$$
\begin{array}{lll}
ty{\downarrow} & \in & [\![ty]\!] \to [\![\langle \mathrm{bitwidth}(ty) \times \mathbf{i}1 \rangle]\!] \\
ty{\uparrow} & \in & [\![\langle \mathrm{bitwidth}(ty) \times \mathbf{i}1 \rangle]\!] \to [\![ty]\!]
\end{array}
$$

$$
\mathbf{i}sz{\downarrow}(v) \text{ or } ty*{\downarrow}(v) = \begin{cases} \lambda\_.\,\mathbf{poison} & \text{if } v = \mathbf{poison} \\ (std) & \text{otherwise} \end{cases}
$$

$$
\langle sz \times ty \rangle{\downarrow}(v) = ty{\downarrow}(v[0]) +\!\!+ \ldots +\!\!+\, ty{\downarrow}(v[sz-1])
$$

$$\mathbf{i}sz\!\uparrow\!(b) \text{ or } ty{*}\!\uparrow\!(b) = \begin{cases}\mathbf{poison} & \text{if } \exists i.\, b[i] = \mathbf{poison}\\ (std) & \text{otherwise}\end{cases}$$
$$\langle sz \times ty\rangle\!\uparrow\!(b) = \langle ty\!\uparrow\!(b_0), \ldots, ty\!\uparrow\!(b_{sz-1})\rangle$$
$$\text{where } b = b_0 +\!\!+ \ldots +\!\!+ b_{sz-1}$$

For base types, $ty\!\downarrow$ transforms poison into the bitvector of all poison bits, and defined values into their standard low-level representation. For vector types, $ty\!\downarrow$ transforms values element-wise, where $+\!\!+$ denotes the bitvector concatenation. Conversely, for base types, $ty\!\uparrow$ transforms bitwise representations with at least one poison bit into poison, and transforms fully defined ones in the standard way. For vector types, $ty\!\uparrow$ transforms bitwise representations element-wise.

Now we give semantics to selected instructions in Figure 5. It shows how each instruction updates the register file $R \in \text{Reg}$ and the memory $M \in \text{Mem}$, denoted $R, M \hookrightarrow R', M'$. The value $[\![op]\!]_R$ of operand op over R is given by:

$$\begin{aligned}[\![r]\!]_R &= R(r) && \text{// register}\\ [\![C]\!]_R &= C && \text{// constant}\\ [\![\mathbf{poison}]\!]_R &= \mathbf{poison} && \text{// poison}\end{aligned}$$

The load operation $\text{Load}(M, p, sz)$ successfully returns the loaded bit representation only if p is a non-poison address pointing to a valid block of bitwidth at least sz in the memory M. The store operation $\text{Store}(M, p, b)$ successfully stores the bit representation b in the memory M and returns the updated memory only if p is a non-poison address pointing to a valid block of bitwidth at least $\text{bitwidth}(b)$.

The rules shown in Figure 5 follow the standard operational semantics notation. For example, the first rule says that the instruction $r = \textbf{freeze}\ \mathbf{i}sz\ op$, if the operand value $[\![op]\!]_R$ is poison, updates the destination register r with an arbitrary value v (i.e., updates the register file R to $R[r \mapsto v]$) leaving the memory M unchanged; and if $[\![op]\!]_R$ is a non-poison value v, it updates the register r with the operand value v.

5. Illustrating the New Semantics

In this section we show how the proposed semantics enable optimizations that cannot be performed soundly today in LLVM. We also show how to encode certain C/C++ idioms in LLVM IR for which changes are required in the front-end (Clang), as well as optimizations that need tweaks to remain sound.

5.1 Loop Unswitching

We showed previously that GVN and loop unswitching could not be used together. With the new semantics, GVN becomes sound, since we chose to trigger UB in case of branch on poison value. Loop unswitching, however, requires a simple change to become correct. When a branch is hoisted out of a loop, the condition needs to be frozen. E.g.,

```
while (c) {
  if (c2) { foo }
  else    { bar }
}
```

is transformed into:

```
if (freeze(c2)) {
  while (c) { foo }
} else {
  while (c) { bar }
}
```

By using the `freeze` instruction, we avoid introducing UB in case `c2` is poison and force a non-deterministic choice between the two loops instead. This is a refinement of the original code, which would trigger UB if `c2` was poison and the loop executed at least once.

Freeze can be avoided if the branch on `c2` is placed in the loop pre-header (since then the loop is guaranteed to execute at least once). The compiler further needs to prove that the branch on `c2` is always reachable (i.e., that all function calls before the "`if (c2)`" statement always return).

5.2 Reverse Predication

In some CPU architectures it is beneficial to compile a `select` instruction into a set of branches rather than a conditional move. We support this transformation using freeze:

```
%x = select %c, %a, %b
```

can be transformed to:

```
  %c2 = freeze %c
  br %c2, %true, %false
true:
  br %merge
false:
  br %merge
merge:
  %x = phi [ %a, %true ], [ %b, %false ]
```

Freeze ensures that no UB is triggered if `%c` is poison. We believe, however, that this kind of transformation may be delayed to lower-level IRs where poison usually does not exist.

5.3 Bit Fields

C and C++ have bit fields in structures. These fields are often packed together to form a single word-sized field (depending on the ABI). Since in our semantics loads of uninitialized data yield poison, and bit-field store operations also require a load (even the first store), extra care is needed to ensure that a store to a bit field does not always yield poison.

Therefore we propose to lower the following C code:

```
mystruct.myfield = foo;
```

into:

```
%val  = load %mystruct
%val2 = freeze %val
```

$(r = \textbf{freeze}\ \textbf{i}sz\ op)$

$$\frac{[\![op]\!]_R = \textbf{poison} \quad v \in \text{Num}(sz)}{R, M \hookrightarrow R[r \mapsto v], M} \qquad \frac{[\![op]\!]_R = v \neq \textbf{poison}}{R, M \hookrightarrow R[r \mapsto v], M}$$

$(r = \textbf{freeze}\ ty\ op)$ for $ty = \langle n \times \textbf{i}sz \rangle$

$$\frac{\begin{array}{c} [\![op]\!]_R = \langle v_0, \ldots, v_{n-1} \rangle \\ \left[\begin{array}{c} \forall i.\ (v_i = \textbf{poison} \land v'_i \in \text{Num}(sz)) \\ \lor\ (v_i = v'_i \neq \textbf{poison}) \end{array}\right] \end{array}}{R, M \hookrightarrow R[r \mapsto \langle v'_0, \ldots, v'_{n-1} \rangle], M}$$

$(r = \textbf{phi}\ ty\ [op_1, L_1],\ \ldots,\ [op_n, L_n])$

$$\frac{[\![op_i]\!]_R = v_i}{R, M \hookrightarrow R[r \mapsto v_i], M}\ \text{(coming from } L_i\text{)}$$

$(r = \textbf{select}\ op,\ ty\ op_1,\ op_2)$

$$\frac{[\![op]\!]_R = \textbf{poison}}{R, M \hookrightarrow R[r \mapsto \textbf{poison}], M} \quad \frac{[\![op]\!]_R = 1 \quad [\![op_1]\!]_R = v_1}{R, M \hookrightarrow R[r \mapsto v_1], M} \quad \frac{[\![op]\!]_R = 0 \quad [\![op_2]\!]_R = v_2}{R, M \hookrightarrow R[r \mapsto v_2], M}$$

$(r = \textbf{and}\ \textbf{i}sz\ op_1,\ op_2)$

$$\frac{[\![op_1]\!]_R = \textbf{poison}}{R, M \hookrightarrow R[r \mapsto \textbf{poison}], M} \quad \frac{[\![op_2]\!]_R = \textbf{poison}}{R, M \hookrightarrow R[r \mapsto \textbf{poison}], M}$$

$$\frac{[\![op_1]\!]_R = v_1 \neq \textbf{poison} \quad [\![op_2]\!]_R = v_2 \neq \textbf{poison}}{R, M \hookrightarrow R[r \mapsto v_1\ \&\ v_2], M}$$

$(r = \textbf{add nsw}\ \textbf{i}sz\ op_1,\ op_2)$

$$\frac{[\![op_1]\!]_R = \textbf{poison}}{R, M \hookrightarrow R[r \mapsto \textbf{poison}], M} \quad \frac{[\![op_2]\!]_R = \textbf{poison}}{R, M \hookrightarrow R[r \mapsto \textbf{poison}], M}$$

$$\frac{[\![op_1]\!]_R = v_1 \quad [\![op_2]\!]_R = v_2 \quad v_1 + v_2 \text{ overflows (signed)}}{R, M \hookrightarrow R[r \mapsto \textbf{poison}], M}$$

$$\frac{[\![op_1]\!]_R = v_1 \quad [\![op_2]\!]_R = v_2 \quad v_1 + v_2 \text{ no signed overflow}}{R, M \hookrightarrow R[r \mapsto v_1 + v_2], M}$$

$(r = \textbf{bitcast}\ ty_1\ op\ \textbf{to}\ ty_2)$

$$\frac{[\![op]\!]_R = v}{R, M \hookrightarrow R[r \mapsto ty_2{\uparrow}(ty_1{\downarrow}(v))], M}$$

$(r = \textbf{load}\ ty, ty{*}\ op)$

$$\frac{\text{Load}(M, [\![op]\!]_R, \text{bitwidth}(ty)) \text{ fails}}{R, M \hookrightarrow \text{UB}} \qquad \frac{\text{Load}(M, [\![op]\!]_R, \text{bitwidth}(ty)) = v}{R, M \hookrightarrow R[r \mapsto ty{\uparrow}(v)], M}$$

$(\textbf{store}\ ty\ op_1, ty{*}\ op)$

$$\frac{\text{Store}(M, [\![op]\!]_R, ty{\downarrow}([\![op_1]\!]_R)) \text{ fails}}{R, M \hookrightarrow \text{UB}} \qquad \frac{\text{Store}(M, [\![op]\!]_R, ty{\downarrow}([\![op_1]\!]_R)) = M'}{R, M \hookrightarrow R, M'}$$

Figure 5. Semantics of selected instructions

```
%val3 = ...combine %val2 and %foo...
store %val3, %mystruct
```

We need to freeze the loaded value, since it might be the first store to the bit field and therefore it might be uninitialized. If the stored value `foo` is poison, this bit field store operation contaminates the adjacent fields when it is combined through bit masking operations. This is fine, however, since if `foo` is poison then UB must have already occurred in the source program and so we can taint the remaining fields.

An alternative way of lowering bit fields is to use vectors or use the structure type. These are superior alternatives, since they allow perfect store-forwarding (no freezes), but currently they are both not well supported by LLVM's backend. E.g., with vectors:

```
%val  = load <32 x i1> %mystruct
%val2 = insertelement %foo, %val, ...
store %val2, %mystruct
```

Here we assume the word size is 32 bits, and therefore we ask LLVM to load a vector of 32 bits instead of loading a whole word. Since our semantics for vectors define that poison is determined per element, a poison bit field cannot contaminate adjacent fields.

5.4 Load Combining and Widening

Sometimes it is profitable to combine or widen loads to align with the word size of a given CPU. However, if the compiler chooses to widen, say, a 16-bit load into a 32-bit load, then care must be taken because the remaining 16 bits may be poison or uninitialized and they should not poison the value the program was originally loading. To solve the problem, we also resort to vector loads, e.g.,

```
%a = load i16, %ptr
```

can be transformed to:

```
%tmp = load <2 x i16>, %ptr
%a = extractelement %tmp, 0
```

As for bit fields, vector loads make it explicit to the compiler that we are loading unrelated values, even though at assembly level it is the still the same load of 32 bits.

5.5 Pitfall 1: Freeze duplication

Duplicating freeze instructions is not allowed, since each freeze instruction may return a different value if the input is poison. For example, this blocks loop sinking optimization (dual of loop invariant code motion). Loop sinking is beneficial if, e.g., a loop is rarely executed. For example, it is not sound to perform the following transformation:

```
x = a / b;
y = freeze(x);
while (...) {
  use(y)
}
```

to:

```
while (...) {
  x = a / b;
  y = freeze(x);
  use(y)
}
```

5.6 Pitfall 2: Semantics of static analyses

Static analyses in LLVM usually return a value that holds only if all of the analyzed values are not poison. For example, if we run the `isKnownToBeAPowerOfTwo` analysis on value "`%x = shl 1, %y`", we get a statement that `%x` will be always a power of two. However, if `%y` is poison, then `%x` will also be poison, and therefore it could take any value, including a non-power-of-two value.

Many LLVM analyses are not sound over-approximations with respect to poison. The main reason is that if poison was taken into account then most analyses would return the worst result (top) most of the time, rendering them useless.

This semantics is generally fine when the result of the analyses are used for expression rewriting, since the original and transformed expressions will yield poison when any of the inputs is poison. However, this is not true when dealing with code movement past control-flow. For example, we would like to hoist the division out of the following loop (assuming `a` is loop invariant):

```
while (c) {
  b = 1 / a;
}
```

If the `isKnownToBeAPowerOfTwo` analysis states that `a` is always a power of two, we are tempted to conclude that hoisting the division is safe since `a` cannot possibly be zero. However, `a` may be poison, and therefore hoisting the division would introduce UB if the loop did not execute.

In summary, there is a trade-off for the semantics of static analysis regarding how they treat poison. LLVM is considering extending APIs of relevant analyses to return up-to results with respect to poison, i.e., the result of an analysis is sound if a set of values is non-poison. Then it is up to the client of the analysis to ensure this is the case if it wants to use the result of the analysis in a way that requires the value to be non-poison (e.g., to hoist instructions that may trigger UB past control-flow).

6. Implementation

We prototyped our new semantics in LLVM 4.0 RC4.[9] We made the following modifications to LLVM, changing a total of 578 lines of code:

- Added a new freeze instruction to the IR and to SelectionDAG (SDAG), and added appropriate translation from IR's freeze into SDAG's freeze and then to MachineInstruction (MI).
- Fixed loop unswitching to freeze the hoisted condition (as described in Section 5.1).
- Fixed several unsound InstCombine (peephole) transformations handling select instructions (e.g., the problems outlined in Section 3.4).
- Added simple transformations to InstCombine to optimize spurious uses of freeze, such as transforming `freeze(freeze(x))` to `freeze(x)` and `freeze(const)` to `const`.

We made a single change to Clang, modifying just one line of code: we changed the lowering of bit field stores to freeze the loaded value (as described in Section 5.3).

Lowering freeze LLVM IR goes through two other intermediate languages before assembly is finally generated. Firstly, LLVM IR is lowered into Selection DAG (SDAG) form, which still represents code in a graph like LLVM IR but where operations may already be target dependent. Secondly, SDAG is lowered into MachineInstruction (MI) through standard instruction selection algorithms, followed by register allocation.

We introduced a freeze operation in SDAG, so a freeze in LLVM IR maps directly into a freeze in SDAG. Additionally, we had to teach type legalization (SDAG level) to handle freeze instructions with operands of illegal type (for the given target ISA). For instruction selection (i.e., when going from SDAG to MI), we convert poison values into pinned undef registers, and freeze operations into register copies. At MI level there is no poison, but instead there are undef registers, which may yield a different value for each use like LLVM IR's undef value. Since taking a copy from an undef register effectively freezes undefinedness (i.e., all uses of the copy observe the same value), we can lower freeze into a register copy.

Optimizations We had to implement a few optimizations to recover some performance regressions we observed in early prototypes. These regressions were due to LLVM optimizers not recognizing the new freeze instruction and conservatively giving up. For example, on x86 it is usually preferable to lower a branch on an and/or operation into a pair of jumps rather than do the and/or operation and then do a single jump. This transformation got blocked if the branch was done on

[9] Code available from `https://github.com/snu-sf/{llvm-freeze, clang-freeze}/tree/pldi`

a frozen and/or operation. We modified CodeGenPrepare (a phase right before lowering IR to SDAG) to support freeze.

For x86, a comparison used only by a conditional branch is usually moved so that it is placed right before the branch, since it is often preferable to repeat the comparison (if needed) than save the result to reuse later. Since freeze instructions cannot be sunk into loops, this transformation is blocked if the branch is over a frozen comparison. We changed CodeGenPrepare to transform "`freeze(icmp %x, const)`" to "`icmp(freeze %x), const`" when deemed profitable. Note that we cannot do this transformation early in the pipeline since it would break some static analyses (like scalar evolution)—the transformed expression is a refinement of the original one.

We changed the inliner to recognize freeze instructions as zero cost, even if they may not always be free. With this change, we avoid changing the behavior of the inliner as much as possible.

Testing the prototype To test the correctness of the prototype, we used the LLVM and Clang test suites. We also used opt-fuzz[10] to exhaustively generate all LLVM functions with three instructions (over 2-bit integer arithmetic) and then we used Alive [17] to validate both individual passes (InstCombine, GVN, Reassociation, and SCCP) and the collection of passes implied by the `-O2` compiler flag. This way we increase confidence that Alive and LLVM agree on the semantics of the IR. This technique was also very useful during the development of the semantics since it enabled us to quickly try out different solutions and check which optimizations would be invalid.

Opportunities for improvement Our prototype leaves room for improvement. For example, we discussed in Section 3.2 that hoisting divisions out of loops is currently disabled in LLVM. We did not attempt to reactivate this optimization.

Some optimizations need to be made freeze-aware. At present they will conservatively fail to optimize since they do not recognize the new instruction. For example, GVN does not yet know how to fold equivalent freeze instructions. We consulted with a GVN expert and we were told it is possible to extend the algorithm to support freeze instructions, with the caveat that for GVN to be sound for freeze it has to replace all uses of a given freeze instruction if it wants to replace one of the uses.

Another possible avenue for improvement is to take advantage of the optimization pipeline order. Right now LLVM transformations have to assume the worst-case scenario: they may be run in any order and for an arbitrary number of times. However, the pipeline is usually mostly fixed for a given file (i.e., optimizations do not usually schedule other optimizations to be run afterward). Therefore, it makes sense to take advantage of this fixed order. In particular, we could determine when no more inter-procedural analyses and transformations will be run, and change the semantics of function arguments and global variables to be non-poison (since we cannot observe if they are). In this way, we could reduce the number of freeze instructions.

The lowering of freeze into assembly is currently suboptimal. Our prototype reserves a register for each poison value within a function (during its live range only). However, in certain architectures it is possible to either reuse an already pinned register that does not change within the function (such as EBP or ESP on x86) or use a constant register (such as R0 on ARM that is usually zero). This strategy would allow us to save up to one register per poison value and therefore reduce register pressure.

Limitations of the prototype Our prototype has a few limitations that make it unsound in theory, even though we did not detect any end-to-end miscompilations. These limitations do not reflect fundamental problems with our proposed semantics, but they require more extensive changes to LLVM than we have performed so far. Also, bear in mind that LLVM was already unsound before our changes, but in ways that are harder to fix.

InstCombine performs a few transformations taking a select instruction and producing arithmetic operations. For example, "`select %c, true, %x`" is transformed into "`or %c, %x`". This transformation is incorrect if `%c` may be poison. A safe version requires freezing `%c` for the `or` operation. Alternatively, we could just remove these transformations, but that would likely require improvements to other parts of the compiler to make them recognize the idiom to produce efficient code (since at the moment the backend and other optimizations may not be expecting this non-canonical code).

Another limitation is related to vectors. We have shown that widening can be done safely by using vector operations. However, LLVM does not yet handle vectors as first-class values, which frequently results in generation of sub-optimal code when vectors are used. Therefore, we did not fix any widening done by LLVM (e.g., in GVN, in Clang's lowering of bit-fields, or in Clang's lowering of certain parameters that require widening by some ABIs).

7. Performance Evaluation

This section evaluates the performance of our prototype in terms of compile time and size and speed of generated code.

7.1 Experimental Setup

Environment We used two machines with different microarchitectures for evaluation. Machine 1 had an Intel Core i7 870 CPU at 2.93 GHz, and Machine 2 had an Intel Core i5 6600 CPU at 3.30 GHz. Both machines had 8 GB of RAM and were running Ubuntu 16.04. To get consistent results, we disabled HyperThreading, SpeedStep, Turbo Boost, and address space layout randomization (ASLR). We used the

[10] `https://github.com/regehr/opt-fuzz`

`cpuset` tool[11] to grant exclusive hardware resources to the benchmark process. Machines were disconnected from the network while running the benchmarks.

Benchmarks We used three benchmarks: SPEC CPU 2006, LLVM Nightly Test (LNT), and five large single-file programs ranging from 7k to 754k lines of code each.[12] SPEC CPU consists of 12 integer-only (CINT) and seven floating-point (CFP) benchmarks (we only consider C/C++ benchmarks). LNT consists of 281 benchmarks with about 1.5 million lines of code in total.

Measurements We measured running time and peak memory consumption of the compiler, running time of compiled programs, and generated object file size.

To estimate compilation and running time, we ran each benchmark three times (except LNT, which we ran five times to cope with shorter running times) and took the median value. To estimate peak memory consumption, we used the `ps` tool and recorded the `rss` and `vsz` columns every 0.02 seconds. To measure object file size, we recorded the size of `.o` files and the number of IR instructions in LLVM bitcode files. All programs were compiled with `-O3` and the comparison was done between our prototype and the version of LLVM/Clang from which we forked.

7.2 Results

Compile time On both machines, compile time was largely unaffected by our changes. Most benchmarks were in the range of $\pm 1\%$. There were a few exceptions with small files, such as the "Shootout nestedloop" benchmark, where compilation time increased by 19% to 29 ms. The reason was that an optimization (jump threading) did not kick in because of not knowing about freeze, which then caused a different set of optimizations to fire in the rest of the pipeline.

Memory consumption For most benchmarks, peak memory consumption was unchanged, and we observed a maximum increase of 2% for bzip2, gzip, and oggenc.

Object code size We observed changes in the range of $\pm 0.5\%$. Freeze instructions represented about 0.04%–0.06% of the total number of IR instructions. The gcc benchmark, however, had 3,993 freeze instructions (0.29% of total), since it contains a large number of bit-field operations.

Run time Change in performance for SPEC CPU 2006 is shown in Figure 6. The results are in the range of $\pm 1.6\%$, with slightly different results on the two machines.

For LNT benchmarks, only 26% had different IR after optimization, and only 82% of those produced different assembly (21% overall resulted in a different binary). Excluding noisy tests, we observed a range of -3% to 2% performance change on machine 1 and -3% to 1.5% on machine 2, except for one case: "Stanford Queens." This test showed a significant speedup (6% on machine 1 and 8% on machine 2) because the introduction of a single freeze instruction caused a change in allocated registers (r13 vs r14). According to the Intel Optimization Reference Manual, the latency and throughput of the LEA instruction is worse with certain registers.[13]

It is normal that run time results fluctuate a bit when a new instruction is added to an IR, since some optimizations and heuristics need to learn how to handle the new instruction. We did only a fraction of the required work, but the results are already reasonable, which shows that the semantics can be deployed incrementally.

8. Related Work

We are not aware of any work directly studying the semantics of undefined behavior of IRs. We instead survey work related to orthogonal improvements and extensions to IRs, on undefined behavior, and on formalization of IRs.

Compiler IRs Most of the past work on compiler IRs has focused on extensions to improve efficiency and enable more complex static analyses. However, most of this work has ignored undefined behavior.

Static single assignment form (SSA [6]) is a functional IR where each variable is assigned only once. This enables (time and space) efficient implementations of sparse static analyses. SSA is now used by most production compilers (for imperative languages).

There have been multiple proposals to extend SSA, including gated SSA [21] and static single information form (SSI [1]), whose goal is to enable efficient path-sensitive analyses. In memory SSA form [20, 25], memory is treated like scalar variables and it is put in SSA form, such that load/store instructions take memory as a parameter. This enables, for example, simple identification of redundant loads and easy movement of memory-related instructions.

More recently, Horn clauses have been proposed as a compiler IR [10], following the trend of recent software verification tools [11].

There have also been proposals to expose parallel constructs in IRs using multiple paradigms, such as π-threads [22], INSPIRE [13], SPIRE [15], and Tapir [23].

Undefined Behavior Undefined behavior is not a topic upon which there is widespread agreement either at the source or IR levels [8, 9, 18]. Several tools have been developed in the past few years to detect execution of undefined behavior in programs, such as IOC [7] and STACK [26].

Hathhorn et al. [12] give a formalization of most undefined behaviors in C. Kang et al. [14] give a formalization of

[11] `https://github.com/lpechacek/cpuset.git`

[12] `http://people.csail.mit.edu/smcc/projects/single-file-programs/` and `https://sqlite.org/2016/sqlite-amalgamation-3140100.zip`

[13] `https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf`, §3.5.1.3 Using LEA

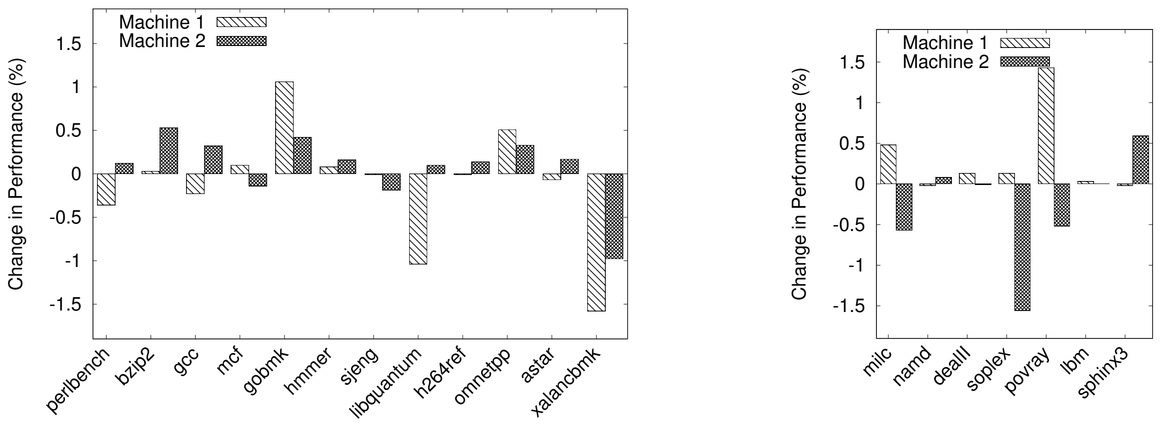


Figure 6. Change in performance in % for SPEC CPU 2006: CINT on the left, CFP on the right. Positive values indicate that performance improved, and negative values indicate that performance degraded.

C's integer to pointer cast, which contains implementation-defined and undefined behavior features.

Formalization of IRs Vellvm [27] is a formalization of parts of the LLVM IR in Coq. It has limited support for undef, and none for poison.

Alive [17] formalizes parts of the LLVM IR as part of its SMT-based VCGen. Alive supports undef and poison values, but has limited support for control-flow manipulating transformations. Souper[14] (a superoptimizer for LLVM) includes limited support for undefined behavior.

CompCert [16] includes full formalization of multiple three-address code-based IRs that are used throughout its pipeline. There is also an extension to CompCert that includes the formalization of an SSA-based IR [3].

Chakraborty and Vafeiadis [5] formalize the semantics of parts of the concurrency-related instructions of LLVM IR.

9. Undefined Behavior in Other Compilers

Most compilers have a concept like LLVM's undef, since it is simple, innocent-looking, and has tangible benefits. There are two common semantics for undef: one where each use of undef may get a different value, as in LLVM and Microsoft Phoenix; and another where all uses of undef get the same value, as in Firm [4], the Microsoft Visual C++ compiler (MSVC), and the Intel C/C++ Compiler (ICC).

GCC attempts to initialize uninitialized variables to zero, or give them a consistent value otherwise.[15] However, this does not appear to be part of GCC's semantics because optimizations like SCCP can assume multiple values for the same uninitialized variable.[16]

Firm additionally has the concept of a "Bad" value,[17] the use of which triggers UB. This semantics is stronger than LLVM's poison (where the use of poison is not necessarily UB; arithmetic operations taking poison as input often just yield poison).

Signed overflow UB is exploited by ICC,[18] MSVC,[19] and GCC.[20] As far as we know, these compilers do not have their semantics formalized, but they appear to use concepts similar to LLVM's poison. At least MSVC seems to suffer from similar problems as the ones we have outlined in this work for LLVM. It is likely that MSVC could fix their IR in a way similar to our solution. Similarly, Firm's developers acknowledge several bugs with their handling of "Bad" values; it is not clear whether it is a fundamental problem with the semantics of their IR or if these are implementation bugs.

CompCert [16] IR also has a deferred UB value called *undef*, which is essentially the same as poison in LLVM. Since branching on undef triggers UB in CompCert, certain optimizations like loop unswitching are unsound and thus not performed by CompCert. Mullen et al. [19] describe how the undef value gets in the way of peephole optimizations in CompCert.

In summary, most modern compiler IRs support reasoning based on undefined behavior, but this reasoning has received little up-front design work or formal attention.

10. Future Work

In this section we describe ongoing and future work to improve the prototype we have built and the path for integration in LLVM, as well as some of the research challenges that remain.

[14] `https://github.com/google/souper`

[15] `https://github.com/gcc-mirror/gcc/blob/ad7b10a21746a784e7e8edeb606cc99cf2853e21/gcc/init-regs.c#L32`

[16] `https://godbolt.org/g/r4PX4A`

[17] `http://pp.ipd.kit.edu/firm/Unknown_and_Undefined`

[18] `https://godbolt.org/g/egCqqm`

[19] `https://godbolt.org/g/ojfRVd`

[20] `https://godbolt.org/g/gtEbXx`

10.1 Implementation

We have submitted an initial set of patches to the LLVM community to (1) introduce the freeze instruction, (2) make it explicit in the documentation that branch on undef/poison is UB, and (3) fix the identified bug in loop unswitching. These patches are currently under review, and we expect the process to take a few more months, since these changes affect the core of the compiler, which is its IR. Following acceptance of this first set of patches, we plan to work with the LLVM community to replace the undef value with poison in an incremental, but safe, fashion. We are also working with other compiler vendors (Microsoft) to evaluate whether a semantics similar to poison/freeze is adequate.

Further work is required to ensure a safe transition to a world without undef. For example, there are several transformations in LLVM which are analogous to load widening, and they can be similarly fixed with bitvector load/stores. For example, small memcpy calls can be optimized into load/store operations of 4 or 8-bytes integers, but this is incorrect under the proposed semantics because existence of a poison bit in an input array element may contaminate the entire loaded value. Also, clang sometimes loads multiple fields of a struct with a single load, which may even include padding bits.

Scalar evolution is an example of an analysis that must learn how to deal with freeze. It computes, for example, how many times a loop will run, and it is used by most loop transformations. Scalar evolution is critical for the performance of programs with loops but it currently fails to analyze expressions involving freeze.

10.2 Research Agenda

This paper leaves many unanswered questions for future work. For example, it is not well understood what is the trade-off between different semantics for deferred undefined behavior (UB)—we proposed a fix, but there are other possibilities—what is the real-life impact of exploiting UB exposed by languages like C and C++ in terms of performance and from where does most of the benefit comes from, or what is the benefit of having UB in the IR for compilers targeting safe languages.

Other constructs that modern compilers' IRs have, such as assumptions, barriers, implicit control-flow, etc, while they seem correct, are easily miscompiled by innocent-looking optimizations. Further research is needed to improve usability and usefulness of these features.

Reassociation is a conceptually simple transformation that rewrites polynomials in some canonical form. In doing so, it usually has to remove overflow assumptions of subexpressions (like `nsw` in LLVM) since reassociation may change how and if subexpressions overflow. However, dropping the `nsw` attribute from expressions inhibits later optimizations (such as induction variable widening). At least LLVM and MSVC have suffered from bugs because of reassociation not dropping overflow assumptions on subexpression and a later optimization took advantage of it, resulting in end-to-end miscompilations. Further research is needed to determine whether this phase ordering issue can be eliminated completely with better IR semantics.

In this work we have only covered the LLVM IR, which is the IR used by LLVM's middle-end optimizers. LLVM, like other compilers, has other lower-level representations (SelectionDAG and MachineInstruction) that are used in later phases (e.g., for instruction selection). For example, SelectionDAG has an undef value and developers are considering whether introducing poison would be sound and useful. Further research is needed in this area of low-level IRs.

Formalizing semantics of an IR to which a compiler adheres opens many avenues for automation. Tools for translation validation or optimization verification, superoptimization, synthesis of optimizations, and so on become now possible. We believe that many parts of a compiler should and can now be developed with the help of automation.

11. Conclusion

Undefined behavior in a compiler IR, which is not necessarily related to undefined behavior in any given source language, gives optimizers the freedom to perform desirable transformations. We have presented the first detailed look at IR-level undefined behavior that we are aware of, and we have described difficult, long-standing problems with the semantics of undefined behavior in LLVM IR. These problems are present to some extent in other modern optimizing compilers. We developed and prototyped a modified semantics for undefined behavior that meets our goals of justifying most of the optimizations that LLVM currently performs, putting the semantics of LLVM IR on firm ground, and not significantly impacting either compile time or quality of generated code.

Acknowledgments

The authors acknowledge the lengthy discussions with and feedback on previous drafts from Andy Ayers, Daniel Berlin, Chandler Carruth, Hal Finkel, Eli Friedman, Dan Gohman, Jeehoon Kang, Richard Smith, and Joseph Tremoulet. This work was partially supported by the Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1502-07 and by the US National Science Foundation under award CNS-1218022.

References

[1] C. S. Ananian. The static single information form. Master's thesis, MIT, 1999.

[2] Atmel Inc. AVR32 architecture document, Apr. 2011. URL `http://www.atmel.com/images/doc32000.pdf`.

[3] G. Barthe, D. Demange, and D. Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM Trans. Program. Lang. Syst.*, 36(1):4:1–4:35, Mar. 2014.

[4] M. Braun, S. Buchwald, and A. Zwinkau. Firm—a graph-based intermediate representation. Technical Report 35, Karlsruhe

Institute of Technology, 2011. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000025470`.

[5] S. Chakraborty and V. Vafeiadis. Formalizing the concurrency semantics of an LLVM fragment. In *CGO*, 2017.

[6] 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. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[7] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *ICSE*, 2012.

[8] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *SPW*, 2015.

[9] M. A. Ertl. What every compiler writer should know about programmers. In *KPS*, 2015.

[10] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Horn clauses as an intermediate representation for program analysis and transformation. *Theory and Practice of Logic Programming*, 15(4-5):526–542, July 2015.

[11] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[12] C. Hathhorn, C. Ellison, and G. Roşu. Defining the undefinedness of C. In *PLDI*, 2015.

[13] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer. INSPIRE: The Insieme parallel intermediate representation. In *PACT*, 2013.

[14] J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis. A formal C memory model supporting integer-pointer casts. In *PLDI*, 2015.

[15] D. Khaldi, P. Jouvelot, F. Irigoin, C. Ancourt, and B. Chapman. LLVM parallel intermediate representation: design and evaluation using OpenSHMEM communications. In *Workshop on the LLVM Compiler Infrastructure in HPC*, 2015.

[16] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[17] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *PLDI*, 2015.

[18] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell. Into the depths of C: Elaborating the de facto standards. In *PLDI*, 2016.

[19] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman. Verified peephole optimizations for CompCert. In *PLDI*, 2016.

[20] D. Novillo. Memory SSA – a unified approach for sparsely representing memory operations. In *Proc. of the GCC Developers' Summit*, 2007.

[21] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI*, 1990.

[22] F. Peschanski. Parallel computing with the pi-calculus. In *DAMP*, 2011.

[23] T. B. Schardl, W. S. Moses, and C. E. Leiserson. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *LCPC*, 2016.

[24] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Kelsey, W. Clinger, J. Rees, R. B. Findler, and J. Matthews. Revised^6 report on the algorithmic language Scheme, Sept. 2007. URL `http://www.r6rs.org/final/r6rs.pdf`.

[25] B. Steensgaard. Sparse functional stores for imperative programs. In *ACM SIGPLAN Workshop on Intermediate Representations*, 1995.

[26] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *SOSP*, 2013.

[27] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, 2012.