Evaluating Value-Wise Poison Values for the LLVM Compiler

Filipe Parrado de Azevedo
filipe.p.azevedo@tecnico.ulisboa.com

INESC-ID/Instituto Superior Técnico, Lisboa, Portugal

April 2020

Abstract

The Intermediate Representation (IR) of a compiler has become an important aspect of optimizing compilers in recent years. The IR of a compiler should make it easy to perform transformations while also giving portability to the compiler. One aspect of IR design is the role of Undefined Behavior (UB). UB is important to reflect the semantics of UB-heavy programming languages, like C and C++, namely allowing multiple desirable optimizations to be made and the modeling of unsafe low-level operations. Consequently, the IR of important compilers, such as LLVM, GCC or Intel's compiler, supports one or more forms of UB.

In this work we focus on the LLVM compiler infrastructure and how it deals with UB in its IR, with the concepts of "poison" and "undef", and how the existence of multiple forms of UB conflict with each other and cause problems to very important "textbook" optimizations, such as some forms of "Global Value Numbering" and "Loop Unswitching", hoisting operations past control-flow, among others.

To solve these problems we introduce a new semantics of UB to the LLVM, explaining how it can solve the different problems stated, while most optimizations currently in LLVM remain sound. Part of the implementation of the new semantics is the introduction of a new type of structure to the LLVM IR – Explicitly Packed Structure type – that represents each field in its own integer type with size equal to that of the field in the source code. Our implementation does not degrade the performance of the compiler.

Keywords: Compilers, Undefined Behavior, Intermediate Representations, Poison Values, LLVM, Bit Fields

1. Introduction

A compiler is a complex piece of computer software that translates code written in one programming language (source language) to another (target language, usually assembly of the machine it is running on). Aside from translating the code, some compilers, called optimizing compilers, also optimize it by resorting to different techniques.

When optimizing code, compilers need to worry about Undefined Behavior (UB). UB refers to the result of executing code whose behavior is not defined by the language specification (document that defines its behaviors) in which the code is written, for the current state of the program, and may cause the system to have a behavior which was not intended by the programmer.

1.1. Motivation

The motivation for this work is the countless bugs that have been found over the years in LLVM[1] due to the contradicting semantics of UB in the LLVM intermediate representation. Since LLVM is used by some of the most important companies in the computer science area, these bugs can have dire consequences in some cases.

[1]Some examples are https://llvm.org/PR21412, https://llvm.org/PR27506, https://llvm.org/PR31652, https://llvm.org/PR31632 and https://llvm.org/PR31633

1.2. Goals

The goal of our work is to implement new UB semantics for the LLVM IR. The current UB semantics diverge between different parts of LLVM and are sometimes contradicting with each other. The PLDI'17 paper [1] proposes semantics that eliminates a form of UB and extends the use of another. These new semantics will be the focus of this report, in which we will describe them, and the benefits and flaws they might have. We will also explain how we implemented them, and the way we measured and evaluated their performance.

2. Background

In this section we present important compiler concepts and some work already done on this topic, as well as current state of LLVM regarding UB.

2.1. Compilers

Optimizing compilers, aside from translating the code between two different programming languages, also optimize it by resorting to different optimization techniques. However, it is often difficult to apply these techniques directly to most source languages, and so the translation of the source code usually passes through intermediate languages [2, 3], that hold more specific information, until it reaches the target language.

These intermediate languages are referred to as Intermediate Representations (IR). Aside from enabling optimizations, the IR also gives portability to the compiler by allowing it to be divided into front-end, middle-end and back-end. The front-end analyzes and transforms the source code into the IR. The middle-end performs CPU architecture independent optimizations on the IR. The back-end is the part responsible for CPU architecture specific optimizations and code generation. This division of the compiler means that we can compile a new programming language by changing only the front-end, and we can compile to the assembly of different CPU architectures by only changing the back-end, while the middle-end and all its optimizations can be shared be every implementation.

Some compilers have multiple IR's, and each one retains and gives priority to different information about the source code that allows different optimizations, which is the case with LLVM. In fact, we can distinguish three different IR's in the LLVM pipeline: the LLVM IR[2],; the SelectionDAG[3]; and the Machine-IR[4].

Optimizing compilers need an IR that facilitates transformations and offers efficient and precise static analyses. To be able to do this, one of the problems optimizing compilers have to face is how to deal with Undefined Behavior (UB), which can be present in the source programming language, in the compiler's IR and in hardware platforms. UB results from the desire to simplify the implementation of a programming language. The implementation can assume that operations that invoke UB never occur in correct program code, making it the responsibility of the programmer to never write such code. This makes some program transformations valid which gives flexibility to the implementation. Furthermore, UB is an important presence in compiler's IR's not only for allowing different optimizations but also as a way for the front-end to pass information about the program to the back-end.

In LLVM, UB falls into two categories: immediate UB and deferred UB. Immediate UB refers to operations whose results can have lasting effects on the system. If the result of an instruction that triggered immediate UB reaches a side-effecting operation, the execution of the program must be halted. This characteristic gives freedom to the compilers to not even emit all the code up until the point where immediate UB would be executed. Deferred UB refers to operations that produce unforeseeable values but are safe to execute otherwise.

Deferred UB is necessary to support speculative execution of a program. Otherwise, transformations that rely on relocating potentially undefined operations would not be possible. The division between immediate and deferred UB is also important because deferred UB allows optimizations that otherwise could not be made. If this distinction was not made, all instances of UB would have to be treated equally and that means treating every UB as immediate UB, i.e., programs cannot execute them since it is the stronger definition of the two.

2.2. Undefined Behavior in Current Optimizing Compilers

The recent scientific works that propose formal definitions and semantics for compilers that we are aware of all support one or more forms of UB.

LLVM

As discussed before, the LLVM IR (just like the IR of many other optimizing compilers) supports two forms of UB that allows it to be more flexible when UB might occur and maybe optimize that behavior away.

Deferred UB comes in two forms in LLVM [1]: an `undef` value and a `poison` value. The `undef` value corresponds to an arbitrary bit pattern for that particular type, i.e., an arbitrary value of the given type, and may return a different value each time it is used. The `undef` (or a similar concept) is also present in other compilers.

The other form of deferred UB in LLVM is the `poison` value, which is a slightly more powerful form of deferred UB than `undef`, and taints the Data-Flow Graph [4, 5], meaning that the result of every operation with `poison` is `poison`. For example, the result of an `and` instruction between `undef` and `0` is `0`, but the result of an `and` instruction between `poison` and `0` is `poison`. This way, when a `poison` value reaches a side-effecting operation, it triggers immediate UB.

Having both forms of deferred UB permits different optimizations. However, the presence of two forms of deferred UB is unsatisfying and the interaction between them has often been a persistent source of discussions and bugs.

[2]https://llvm.org/docs/LangRef.html

[3]https://llvm.org/docs/CodeGenerator.htmlintroduction-to-selectiondags

[4]https://llvm.org/docs/MIRLangRef.html

CompCert

CompCert, introduced in [6], is a formally verified, realistic compiler, developed using the Coq proof assistant [7]. CompCert holds proof of semantic preservation, meaning that the generated machine code behaves as specified by the semantics of the source program.

Behaviors reflect accurately what the outside world the program interacts with can observe. The behaviors we observe in CompCert include *termination*, *divergence*, *reactive divergence*, and *"going wrong"*[5]. Termination means that the compiled code has the same behavior of the source code, with a finite trace of observable events and an integer value that stands for the process exit code. Divergence means the program runs on forever (like being stuck in an infinite loop) with a finite trace of observable events, without doing any I/O. Reactive divergence means that the program runs on forever with an infinite trace of observable events, infinitely performing I/O operations separated by small amounts of internal computations. Finally, "going wrong" behavior means the program terminates but with an error, by running into UB, with a finite trace of observable events performed before the program gets stuck.

Unlike LLVM, CompCert does not have the `undef` value nor the `poison` value to represent Undefined Behavior, using instead "going wrong" to represent every UB, which means that it does not exist any distinction between immediate and deferred UB. This is because the source language, Clight, is deterministic and specified the majority of the sources of UB in C, and the ones that Clight did not specify are serious errors that can have devastating side-effects for the system and should be immediate UB.

Vellvm

The Vellvm (verified LLVM) introduced in [8] is a framework that includes formal semantics for LLVM and associated tools for mechanized verification of LLVM IR code, IR to IR transformations, and analyses, built using the Coq proof assistant, just like CompCert. But, unlike the CompCert compiler, Vellvm has a type of deferred Undefined Behavior semantics (which makes sense since Vellvm is a verification of LLVM): the `undef` value.

This form of deferred UB of Vellvm, though, returns the same value for all uses of a given `undef`, which differs from the semantics of the LLVM. The presence of this particular semantics for `undef`, however, creates a significant challenge when verifying the compiler - being able to adequately capture the non determinism that originates from `undef` and its intentional under-specification of certain incorrect behaviors. Vellvm doesn't have a `poison` value which means that it suffers from the same problems that LLVM has without it - some important textbook transformations are not allowed because using `undef` as the only semantics for UB does not cover every transformation when it comes to potentially undefined operations.

Concurrent LLVM Model

With the arrival of the multicore era, programming languages introduced first-class platform independent support for concurrent programming. LLVM had to adapt to these changes with a concurrency model of its own to determine how the various concurrency primitives should be compiled and optimized. The work by [9] proposes a formal definition of the concurrency model of LLVM, how it differs from the C11 model and what optimizations the model enables.

A concurrency model is a set of premises that a compiler has to fulfill and that programmers can rely upon [9]. The LLVM compiler follows the concurrency model of C/C++ 2011, in which a data race between two writes results in UB, but with a crucial difference: while in C/C++ a data race between a non-atomic read and a write is declared to be immediate UB, in LLVM such a race has defined behavior - the read returns an `undef` value. Despite being a small change, this has a profound impact in the program transformations that are allowed.

2.3. Problems with LLVM and Basis for this Work

As we discussed in the previous section, the presence of two kinds of deferred Undefined Behavior is the cause of inconsistencies in the compilation of programs in LLVM. In this section we will take a look at these inconsistencies and why they exist.

2.3.1 Benefits of Poison

Some optimizations are not correct with only one form of UB in the IR. Suppose we have the following code: `a + b > a`. We can easily conclude that a legal optimization is `b > 0`. Now suppose that `a = INT_MAX` and `b = 1`. In this case, `a + b` would overflow, returning `undef`, and `a + b > a` would return false (since `undef` is an arbitrary value of the given type and there is no integer value greater than `INT_MAX`), while `b > 0` would return true. This means that the semantics of the program were changed, making this transformation illegal.

The `poison` value solves these types of problems. Suppose that `a = INT_MAX` and `b = 1`, just like before. But now, overflow results in a `poison` value.

[5]http://compcert.inria.fr/doc/html/compcert.common.-Behaviors.html

This means that `a + b > a` evaluates to `poison` and `b > 0` returns true. In this case, we are refining the semantics of the program by optimizing away the Undefined Behavior, which makes the optimization legal.

To be able to perform this optimization (among others) the `poison` values were introduced. Although this solved some optimizations, like the two previous cases we have observed, there are many more that are also inconsistent with the semantics of `poison`. Aside from that, some optimizations that `poison` values and `undef` values provide become inconsistent when both types of deferred UB are present.

2.3.2 Loop Unswitching and Global Value Numbering Conflicts

Loop unswitching is an optimization that consists in switching the conditional branches and the loops, if the if-statement condition is loop invariant, as in the following example:

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

to

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

For loop unswitching to be sound, branching on `poison` cannot be UB, because then we would be introducing UB if `c2` was `poison` and `c` was false.

Global value numbering (GVN) [10] corresponds to finding equivalent expressions and then pick a representative and remove the remaining redundant computations. For example in the following code:

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

if we apply GVN, the resulting code would be:

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

Consider now that `y` is `poison` and `w` is not. If the semantics say that branching on `poison` is not UB, but simply a non-deterministic choice, as we did for loop unswitching, in the original program we would not have UB but in the optimized version we would pass a `poison` value as a parameter to the function. However, if we decide that branching on `poison` is UB, then loop unswitching will become unsound while GVN becomes a sound transformation, since the comparison `t == y` would be `poison` and therefore the original program would already be executing UB.

In other words, loop unswitching and GVN require conflicting semantics for branching on `poison` in the LLVM IR to become correct. Hence, by assuming conflicting semantics they perform conflicting optimizations, which enables end-to-end miscompilations.

2.3.3 Select and the Choice of Undefined Behavior

The `select` instruction, which, just like the ternary operation `?:` in C, uses a Boolean to choose between its arguments, is another case where the conflicting semantics of UB in LLVM are apparent. The choice to produce `poison` if any of the inputs is `poison`, or just if the value chosen is `poison` can be the basis for a correct compiler, but LLVM has not consistently implemented either one. The *SimplifyCFG* pass performs a transformation of conditional branches to `select` instructions, but for this transformation to be correct `select` on `poison` cannot be UB if branch on `poison` is not. Sometimes LLVM performs the reverse transformation, and for that case to be correct, branch on `poison` can only be UB if `select` on a `poison` condition is UB.

Since we want both transformations to be achievable, we can conclude that `select` on `poison` and branching on `poison` need to have the same behavior.

2.3.4 Bit Fields and Load Widening

Another problem `poison` creates is accessing bit fields. Some programming languages, such as C/C++, allow bit fields in their structures, that is, a bit or group of bits that can be addressed individually but that are usually made to fit in the same word-sized field.

```
struct {
  unsigned int a : 3;
  unsigned int b : 4;
} s;
```

In this example, we can observe that both variables `a` and `b` fit in the same 32-bit word. While this method saves space, if one of those fields is `poison`, then the other bit will become `poison` as well if we access either one, since they are both stored in the same word. Since every `store` to a bit field requires a `load` to be done first, because the shortest `store` that can be done is the size of the bit width of the type, we need to `load` the entire word, perform the operation needed and then combine the different fields and store them. Since a `load` of an uninitialized position returns `poison`, if we are not careful, the first `store` to a bit field will always result in `poison` being stored.

Another complication that `poison` generates is about load combining/widening. Sometimes it is useful to combine or widen the loads to make them more efficient. For example, if the size of a word in a given processor is 32 bits, and we want to load a 16-bit value, it is often useful to load the entire 32-bit word at once. If we do not take care, however, we might end up "poisoning" both values if one of them is `poison`.

3. Semantics

In the previous section we showed that the current state of UB in LLVM is unsatisfactory, in the sense that a considerable part of optimizations that should be made possible by representing UB in the IR are actually unsound for many cases.

The solution proposed by [1] to resolve these issues was:

- Discard `undef` and only use `poison`.
- Introduce a new instruction `freeze` that non-deterministically chooses a value if the input is `poison`, and is a `nop` otherwise.
- All operations over `poison` return `poison` except `freeze`, `select` and `phi`.
- Branching on `poison` is immediate UB.

The `freeze` instruction was already created as patches[6][7][8] to the LLVM by the authors of [1].

The choice to eliminate either `poison` or `undef` was made because the presence of both forms of UB created more problems than the ones it solved. According to [1], `phi` and `select` were made to conditionally return `poison` because it reduces the amount of `freeze` instructions that had to be implemented. Defining branch on `poison` to be UB enables analyses to assume that the conditions used on branches hold true inside the target block (e.g., when we have `if(x > 0) { ... }` we want to be able to assume that inside the `if` block, `x` is greater than `0`).

One problem with the use of the `freeze` instruction though is that it disables further optimizations that take advantage of `poison`.

[6]https://reviews.llvm.org/D29011
[7]https://reviews.llvm.org/D29014
[8]https://reviews.llvm.org/D29013

3.1. Semantics

The paper by [1] defines the semantic domain of LLVM as follows:

$$\begin{array}{lcl}
\mathrm{Num}(sz) & ::= & \{ i \mid 0 \le i < 2^{sz} \} \\
\mathbf{i}sz & ::= & \mathrm{Num}(sz) \uplus \{ \mathbf{poison} \} \\
ty{*} & ::= & \mathrm{Num}(32) \uplus \{ \mathbf{poison} \} \\
\langle sz{\times}ty \rangle & ::= & \{0, \ldots, sz-1\} \to ty \\
\mathrm{Mem} & ::= & \mathrm{Num}(32) \not\to \langle 8{\times}\mathbf{i}1 \rangle \\
\mathrm{Name} & ::= & \{\%\mathtt{x},\ \%\mathtt{y}\ , \ldots \} \\
\mathrm{Reg} & ::= & \mathrm{Name} \to \{ (ty, v) \mid v \in ty \}
\end{array}$$

Here, Num(sz) refers to any value between 0 and 2^{sz}, where sz refers to the bit width of the value. $\mathbf{i}sz$ refers to the set of values of bit width sz or `poison` (disjoint union). ty corresponds do the set of values of type ty, which can be either `poison` or fully defined value of base types, or element-wise defined for vector types. $ty*$ denotes the set of memory addresses (we assume that each address has a bit width of 32 for simplicity). $\langle sz \times ty \rangle$ is a function representing a vector of sz elements, each one of type ty, meaning that the vector itself is of type ty. The memory Mem is a partial function and it maps a 32 bit address to a byte (partial because not every address is allocated at a given instance). Name alludes to the space of names fit to be a variable name. And finally, the register file Reg corresponds to a function that maps the name of a variable to a type and a value of that type.

The new semantics for selected instructions are defined in Figure 1, where they follow the standard operational semantics notation. It shows how each instruction updates the register file R $\in$ Reg and memory M $\in$ Mem, in the form R, M $\hookrightarrow$ R', M'. The value op_R of operand op over register R is given by: $r_R = R(r)$, for a register; $C_R = C$, for a constant; and $\mathbf{poison}_R = \mathbf{poison}$, for `poison`.

In Figure 1, there can also be seen two meta operations, used to define the semantics of instructions: $ty\!\downarrow$ and $ty\!\uparrow$. $ty\!\downarrow$ transforms poisonous base types into a bitvector of all `poison` bits, and into their standard low-level representation, otherwise. On the other hand, $ty\!\uparrow$ transforms base types bitvectors with at least one `poison` bit into `poison`, and non-poisonous bitvectors into the respective base type value. For vector types, $ty\!\uparrow$ and $ty\!\downarrow$ transform the values element-wise.

3.2. Illustrating the New Semantics

In this section, we will see how the new proposed semantics deals with those problems and what new problems arise by eliminating the `undef` values.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 1: Semantics of selected instructions [1].

3.2.1 Loop Unswitching and GVN

It was previously showed that Loop Unswitching and GVN required conflicting UB semantics for both optimizations to be correct, making it impossible for them to be sound simultaneously. With the introduction of the `freeze` instruction this is no longer the case. The new semantics say that branch on a `poison` value is UB, making the GVN optimization sound, while loop unswitching becomes unsound. However, `freeze` can be used to effectively "freeze" the value of the conditional branch, which would be the cause of UB in case the loop unswitching optimization was made.

3.2.2 Select

As was said before, the `select` instruction is similar to the ternary operator `?:`, in C. In some CPU architectures, it is beneficial to transform a `select` instruction into a series of branches. This transformation is made correct by "freezing" the condition of the `select`.

3.2.3 Bit Fields

As was addressed before, some programming languages allow bit fields in their structures: a bit or group of bits that can be addressed individually but that are usually made to fit in the same word-sized field.

Since every `store` to a bit field requires a `load` to be done first, because the shortest `store` that can be done is the size of the bit width of the type, we need to `load` the entire word, perform the operation needed and then combine the different fields and store them.

Our proposed solution is to create a new type of structure in the LLVM IR where each bit field is stored in its own word. As an example, the current IR of a structure `s` with two integer bit fields, sizes 3 and 4, would be:

```
%struct.s = type { i8, [3xi8] }
```

while the new structure would be represented by:

```
%struct.s = type { i3, i4, i1, [3xi8] }
```

where the last 1-bit word corresponds to padding so that the final size of the word where the bit fields are stored is a multiple of 8 bits, and the array of 3 8-bit elements is another padding introduced to bring the total size of the structure to 32 bits, which is the space the type of the field in the structure, in this case an integer, would occupy.

By expressing the structures this way in the IR, the bit-wise operations that are needed to access bit fields would not need to be emitted here. What we are doing with this solution is to delay the emission of all the bit field operations, and emitting them further down the pipeline, in the next IR - the SelectionDAG - where exists much less UB. Although we try to optimize away `poison` in the LLVM IR, it still exists in the SelectionDAG, giving our solution to stop the spreading of `poison` between bit fields in the LLVM IR only a temporary status, in the sense that this same bit field problem can appear later in the pipeline. In theory we can propagate this new bit field type to the SelectionDAG, getting rid of the problem completely, as the next IR in the LLVM pipeline - the MachineIR - does not contain poison.

3.2.4 Load Combining and Widening

We previously discussed the cautions we must have to not "poison" adjacent values when combining or widening loads. To deal with this, load combining and widening is lowered using vector loads. In a CPU architecture where a word is 32 bits, instead of just loading 16 bits:

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

we could load the entire 32 bit word with a vector type, like this:

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

3.3. Cautions to have with Freeze

There are some problems that arise when we take the freeze instruction into account.

The first one is that the duplication of freeze instructions should not be allowed. Since freeze may return a different value every time it is used, if its input is poison we cannot do some optimizations that rely on sinking instructions into a loop, for instance, which can be helpful when the loop is rarely executed.

Another problem comes from static analyses of programs. In LLVM, static analyses return a value that only holds if none of the analyzed values are poison. Static analyses do not take poison into account because in the case where one of the possible values is poison, the analysis returns the worst case scenario - a poison value - therefore making it useless. This is not a problem when the analysis are used for expression rewriting because in that case both the original and transformed expressions will return poison if any of its inputs is poison. However, if we use the analysis to hoist an expression past control-flow, for example, if it does not take poison into account and it says that the expression is safe to hoist, then we might be introducing UB into the program.

4. Implementation

We started the implementation of the solution by taking care of poison in the bit fields. That task, however, proved to be complex enough to be in a document of its own.

4.1. The Explicitly Packed Structure Solution

The goal of this new type was to represent each bit field separately, delaying the emission of the bit wise operations needed to access bit fields to the next intermediate representation in the pipeline (SelectionDAG). In this new representation we separate the bit fields, associating each with its own word (in the LLVM IR), eliminating the worry of propagating poison to the other bit fields if any of them was poison, since now each load or store only targets an individual bit field. We also decided to calculate the padding (if needed) in between words and insert it in the array that contained the types of the fields (hence the name Explicitly Packed: the structure is already packed with padding, even though the structure might not be a packed structure). This padding in between words refers, for example, to the padding of 3 bytes after a char field so that every field is stored in a multiple of 4 address. We use slashes to denote that a particular structure has explicit packing.

So if we have the following structure in the C programming language:

```
struct {
  char c : 2;
  int i : 2;
}
```

The corresponding LLVM IR will become:

```
%s = type \{ i2, i2, i4, [3 x i8] }/
```

Instead of:

```
%s = type { i8, [3 x i8] }
```

With this new type of structure, if we want to store a 1 into the integer field i, the new LLVM IR will be:

```
%1 = i2* getelementptr %s* @s, 0
store i2 1, i2* %1
```

where in the old LLVM IR (current representation used by Clang) we have:

```
%1 = i8* getelementptr %s* @str, 0
%2 = load i8, i8* %1
%3 = and i8 %2, -13
%4 = or i8 %3, 4
store i8 %4, i8* %1
```

where @str is the name of the variable of the structure type. The getelementptr instruction returns the address of a subelement of an aggregate data structure (arrays, vectors and structures), meaning that it only performs address calculation and does not access memory, as opposed to the load and store instructions. The other instructions are self explanatory.

The current representation used by Clang emits to the IR the bit wise operations, while our new implementation doesn't. Of course this does not mean that we have less instructions in the end, it just means that the bit arithmetic that was previously done in the IR is not there anymore. However, it still needs to be emitted. We now emit those instructions in the SelectionDAG.

4.2. Benefits of the Explicitly Packed Structure

The main benefit of this new representation is to stop of propagation of `poison` to the adjacent bit fields. Aside from that, the Explicitly Packed Structure type gives readability to the IR code by representing the bit fields like the C programming language, while also telling the programmer the exact size the structure will have in memory by showing in the IR type the entire padding.

One major difference is the size of the IR code and the optimizations that it entails. This new type delays the emission of the bit wise operations to the SelectionDAG, meaning that the size of the IR will usually be smaller than the size of the current representation used by Clang. However, if we are storing or loading multiple adjacent bit fields the previous IR could emit less instructions since only one store and load was needed for adjacent bit fields. This difference will affect some optimizations that take the size of the code into account to decide when to fire, as is the case with Inlining, an optimization that replaces the call site of a function with the body of that same function.

There are other optimizations that will benefit from this representation: ones that depend on analysis of the program and can more easily track the values of adjacent bit fields, since they are now separate as opposed to being bundled in a single word.

As a last note, by changing the way a type is represented and used across all the different parts of LLVM means that the community will have to understand and get used to this new IR, since it affects all of the LLVM pipeline.

5. Results

We tested our implementation with the LLVM Nightly test-suite. The LLVM Nightly test-suite is a test suite that contains thousands of different benchmarks and test programs. Unfortunately, in our case, only the tests that contained bit fields were relevant to measure, which brought our number of tests down to 121. From this 121 tests, 113 were single source micro-benchmark tests, designed to mostly test the correctness of the compiler, and 8 were multi-source benchmarks and applications, more important to test the performance of the compiler.

Aside from these tests we also decided to use version 4.0.0 of the GCC compiler as a benchmark, since we had access to its single file source code[9]. This GCC file has over 754k lines of C code and over 2700 structures that contain bit fields, making it arguably the most important benchmark.

[9]https://people.csail.mit.edu/smcc/projects/single-file-programs/

5.1. Experimental Setup

To evaluate our compiler we measured running time and peak memory consumption, running time of compiled programs, and generated object file size. In addition we also measured number of instructions in the LLVM IR of the programs.

To estimate compilation and running time, we ran each test three 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 -O0 and -O3 and the comparison was done between our prototype and the version of LLVM/Clang from which we forked.

We disabled Fast Instruction Selection for the `getelementptr` instruction, and the optimizations SCCP (Sparse Conditional Constant Propagation) and GVN (Global Value Numbering). We swapped the SROA (Scalar Replacement of Aggregates) for the Mem2Reg optimization and disabled the InstCombine Optimization in our implementation for functions that contained access to bit fields.

These changes were made because these algorithms cannot yet recognize the Explicitly Packed Structure, leading to wrong code being generated. Even with these changes, there were some tests that did not pass with the -O3 flag, bringing the total number of tests down to 96, from 122 (including the GCC single file program). Of the 96 remaining programs, only 3 were regular benchmarks while the other 93 were micro-benchmarks.

The machine we used to run the tests on had an Intel Xeon CPU at 2.40GHz, 86.3GB of RAM and was running CentOS Linux 8.

5.2. Compile Time

Compile time was largely unaffected by our changes, either with the -O0 or the -O3 flag. Most benchmarks were in the range of $\pm 2\%$.

With the micro-benchmarks any small change in the tests would equate to a big difference that does not represent the reality. In micro-benchmarks that took more than 5 seconds to compile we observed a maximum change in $+6\%$.

The results with the -O3 flag were identical to the ones with -O0. The benchmarks were in the range of $\pm 1\%$. The micro-benchmarks saw a -10% maximum difference.

5.3. Memory Consumption

For all benchmarks with the -O0 flag peak memory consumption was unchanged, both RSS (Resident Set Size) and VSZ (Virtual Memory Size), all within the $\pm 1\%$ range. The micro-benchmarks saw a RSS value fluctuating between $\pm 4\%$ while the VSZ value maintained values in the $\pm 0.4\%$.

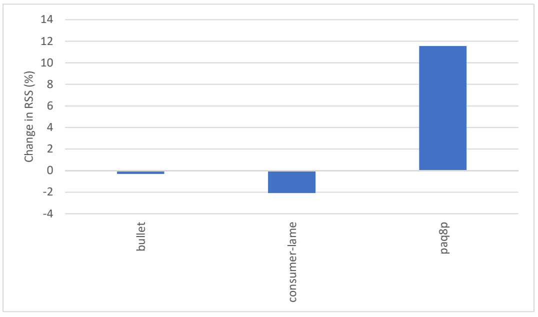

Figure 2: RSS value changes in benchmarks with the -O3 flag.

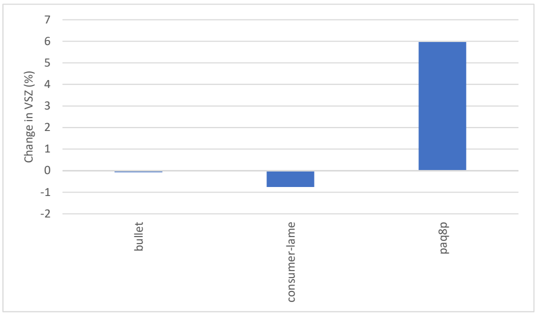

Figure 3: VSZ value changes in benchmarks with the -O3 flag.

Regarding the results with the -O3 flag, the peak memory consumption for the benchmarks kept a ±2% range with a single exception, a test called "paq8p" that saw a significant increase to 11% in the RSS value and 6% in the VSZ value, as shown in Figures 2 and 3. We verified that this difference in peak memory consumption remained even when we transformed the bit fields of the program into regular structure fields. This indicates that the problem is in one of the classes of the LLVM source code, where new fields and functions were introduced to accommodate the Explicitly Packed Struct type. This test generates millions of instances of different structures, which might mean that other tests with these conditions might see a similar rise in peak memory consumption.

On the other hand, the micro-benchmarks stayed unchanged.

5.4. Object Code Size

We measured the size of .o files and the number of IR instructions in the LLVM bitcode files to estimate the size of the object code.

Regarding the size of the .o files compiled with the -O0 flag most benchmarks were unchanged : only GCC and the benchmark "consumer-typeset" were smaller than the original by about 0.71% and 0.75%, respectively. The micro-benchmarks were also mostly unchanged with a maximum increase of 1.37% in the size of .o file for the micro-benchmark "GCC-C-execute-pr70602".

When compiling with the -O3 flag, only the benchmark "bullet" saw an increase of the original by 0.37% while the rest of the benchmarks stayed identical. The micro-benchmarks also remained mostly unchanged with a variation of ±1.6% with the exception of the "GCC-C-execute-990326-1" micro-benchmark which saw an increase of 31% compared to the original. The reason for this outlier is that this benchmark relied extensively on the InstCombine and the subsequent optimizations to reduce the code.

About the number of instructions in the LLVM bitcode file, there was no benchmark/micro-benchmark with a number of instructions superior to their original counter-parts, when compiling with the -O0 flag..

When compiling with the -O3 flag to enable optimizations, the benchmarks remained mostly unchanged, with a maximum increase of 2% for the "bullet" benchmark. However, most of the micro-benchmarks experienced an increase in number of IR instructions, to a maximum of 3000%, simply because of the aforementioned not firing of the InstCombine and subsequent optimizations.

5.5. Run Time

The run time performance was mostly unchanged for benchmarks compiled with the -O0 flag with a maximum decrease in run time of 2%. The compilation with -O3 flag however saw an increase in one of the tests by 4.7%. The increase can be explained by the lack of optimizations after the InstCombine disabling.

5.6. Differences in Generated Assembly

Aside from the measurements taken, we think that it is also important to discuss the differences in the generated assembly, even though the compiled programs have the same behavior when ran.

We witnessed two major differences when comparing the assembly: the code generated when accessing some bit fields, and the extra number of spills and reloads our implementation produced when compared to the LLVM/Clang from which we forked, especially in large programs like GCC and bullet.

The rise in spills and reloads is a consequence of the fact that the heuristic that is in charge of register and memory allocation is not familiar with our implementation.

The reason to why the code to access the bit fields is different is quite simple: even though the `load` and `store` instructions are the only bit field accessing operations that continue present in the LLVM IR, these too were changed. Now they only need to target integers with the size of the actual bit field, and not a whole word. So when the nodes of the bit field accessing instructions in the SelectionDAG are created, we decided to only load or store the minimum amount of bits necessary.

6. Conclusions

The representation of Undefined Behavior (UB) in a compiler has become a very important topic in compiler design. In this work we discuss the pros and cons of having two types of UB representation in the IR and present new semantics to solve these problems, introduced by [1]. In this new semantics we propose to eliminate the `undef` keyword and expand the use of `poison` while also introducing a new instruction, `freeze`, that can simulate the use of `undef`, by "freezing" a `poison` value. This provides a solution to the problems identified with the current representation of UB in the LLVM compiler.

We introduced and implemented a new type of structure type in the LLVM IR - the **Explicitly Packed Structure**. This new type represents each field of the structure in its own integer with size equal to that of the field. Aside from that, it also shows the padding and packing (if needed) that would eventually appear in the assembly code, directly in the structure type in the IR, while not impacting significantly the performance and generated code of the compiler.

6.1. Future Work

As future work is concerned our solution to the problem of propagating `poison` to the adjacent bit fields is not yet finished. Firstly, and as was mentioned before, the new Explicitly Packed Structure type only stops the propagation in the LLVM IR, meaning that the same problem still exists in the next intermediate representation - the SelectionDAG. So a reproduction of this solution in the SelectionDAG is still needed. Furthermore, there are still optimizations that do not take our implementation into account, which needs to be addressed.

Finally, the rest of the semantics still need to be implemented as we only worked on part of the overall `poison` value implementation. Loop Unswitching, GVN and Select have to be updated and other optimizations need to be aware of this new semantics to be able to optimize the new IR, which is a crucial aspect of an optimizing compiler.

References

[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," *SIGPLAN Not.*, vol. 52, no. 6, pp. 633–647, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3140587.3062343

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition).* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[3] J.-P. Tremblay and P. G. Sorenson, *Theory and Practice of Compiler Writing*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1985.

[4] J. Stanier and D. Watson, "Intermediate Representations in Imperative Compilers: A Survey," *ACM Comput. Surv.*, vol. 45, no. 3, pp. 26:1–26:27, Jul. 2013. [Online]. Available: http://doi.acm.org/10.1145/2480741.2480743

[5] J. B. Dennis, "Data Flow Supercomputers," *Computer*, vol. 13, no. 11, pp. 48–56, Nov. 1980. [Online]. Available: http://dx.doi.org/10.1109/MC.1980.1653418

[6] X. Leroy, "Formal Verification of a Realistic Compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1538788.1538814

[7] Y. Bertot and P. Castran, *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[8] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the LLVM Intermediate Representation for Verified Program Transformations," *SIGPLAN Not.*, vol. 47, no. 1, pp. 427–440, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2103621.2103709

[9] S. Chakraborty and V. Vafeiadis, "Formalizing the Concurrency Semantics of an LLVM Fragment," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 100–110. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049832.3049844

[10] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global Value Numbers and Redundant Computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: ACM, 1988, pp. 12–27. [Online]. Available: http://doi.acm.org/10.1145/73560.73562