TÉCNICO LISBOA

Evaluating Value-Wise Poison Values for the LLVM Compiler

Filipe Parrado de Azevedo

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. José Carlos Alves Pereira Monteiro
Dr. Nuno Claudino Pereira Lopes

Examination Committee

Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. José Carlos Alves Pereira Monteiro
Member of the Committee: Prof. Alexandre Paulo Lourenço Francisco

April 2020

Acknowledgments

Firstly, I would like to thank my two dissertation supervisors, Professor José Monteiro and Nuno Lopes, for supporting, helping and guiding me in the right direction on the most important project of my life to date. A compiler is a massive program that I wasn't really ready to tackle and without their knowledge and experience this thesis would not have been possible. Knowing that they cared and kept an eye on me motivated me to work harder than I would have otherwise.

I also would like to thank my father, step-mother and sister for all of their love and support and for putting up with me all these years. Thank you for fully raising me in what is probably the hardest age to raise a person, and for your support after the most traumatic event I experienced.

I want to thank my mother, for raising me and for always being by my side and loving me in the first 15 years of my life. The memories I have of her are plenty and beautiful, she shaped me to be the man I am today and I would not be here without her. Thank you.

I am also grateful to the rest of my family for their continuous support and insight on life, with a special appreciation to the friendship of all of my cousins, who are sometimes harsh but mostly sources of laughter and joy.

Lastly, but definitely not least, I want to extend my gratification to all the colleagues and friends I made over the years, for helping me across different times in my life. Particularly the friends I made over the last 6 years at IST that helped me get through college with their support and encouragement and for mentally helping me grow as a person, with a special gratitude to Ana, Catarina and Martim for putting up with me.

To all of you, and probably someone else I am not remembering – Thank you.

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.

Resumo

A Representação Intermédia (IR) de um compilador tem-se tornado num aspeto importante dos chamados compiladores optimizadores nos últimos anos. A IR de um compilador deve facilitar a realização de transformações ao código e dar portabilidade ao compilador. Um aspeto do design de uma IR é a função do Comportamento Indefinido (UB). O UB é importante para refletir as semânticas de linguagens de programação com muitos casos de UB, como é o caso das linguagens C e C++, mas também porque permite a realização de múltiplas optimizações desejadas e a modelação de operações de baixo nível pouco seguras. Consequentemente, o UB de compiladores importantes, como o LLVM, GCC ou o compilador da Intel, suportam uma ou mais formas de UB.

Neste trabalho o nosso foco é no compilador LLVM e em como é que esta infra-estrutura lida com UB na sua IR, através de conceitos como “poison” e “undef”, e como é que a existência de múltiplas formas de UB entram em conflito entre si e causam problemas a optimizações “textbook” muito importantes, tais como “Global Value Numbering” e “Loop Unswitching”, puxar operações para fora de fluxo de controlo, entre outras.

Para resolver estes problemas introduzimos uma nova semântica de UB no LLVM, explicando como é que esta trata dos problemas mencionados, enquanto mantem as optimizações atualmente no LLVM corretas. Uma parte da implementação desta nova semântica é a introdução de um novo tipo de estrutura na IR do LLVM – o tipo Explicitly Packed Struct – que representa cada campo da estrutura no seu próprio tipo inteiro com tamanho igual ao do seu campo no código de origem. A nossa implementação não degrada o desempenho do compilador.

Palavras Chave

Compiladores; Comportamento Indefinido; Representações Intermédias; Valores Poison; LLVM; Bit Fields

Contents

List of Figures

List of Tables

Acronyms

UB	Undefined Behavior
IR	Intermediate Representation
PHP	PHP: Hypertext Preprocessor
ALGOL	ALGOrithmic Language
PLDI	Programming Language Design and Implementation
CPU	Central Processing Unit
SelectionDAG	Selection Directed Acyclic Graph
SSA	Static Single Assignment
SSI	Static Single Information
GSA	Gated Single Assignment
ABI	Application Binary Interface
GVN	Global Value Numbering
SimplifyCFG	Simplify Control-Flow Graph
GCC	GNU Compiler Collection
SCCP	Sparse Conditional Constant Propagation
SROA	Scalar Replacement of Aggregates
InstCombine	Instruction Combining
Mem2Reg	Memory to Register
CentOS	Community Enterprise Operating System

RSS	Resident Set Size
VSZ	Virtual Memory Size

1

Introduction

Contents

A computer is a system that can be instructed to execute a sequence of operations. We write these instructions in a programming language to form a program. A programming language is a language defined by a set of instructions that can be ran by a computer and, during the last 70 years, these languages have evolved to abstract themselves from its details, to be easier to use. These are called high-level programming languages, and examples are the C, Java and Python languages.

However, a computer can only understand instructions written in binary code and usually the high-level programming languages use natural language elements. To be able to connect these two puzzle pieces we need the help of a specific program - the **compiler**.

1.1 Motivation

A programming language specification is a document that defines its behaviors, and is an important asset to have when implementing or using that same language. Despite being important, it's not obligatory to have a specification and, in fact, some programming languages do not have one and are still widely popular (PHP only got a specification after 20 years, before that the language was specified by what the interpreter did). Nowadays, when creating a programming language, the implementation and the specification are developed together since the specification defines the behavior of a program and the implementation checks if that specification is possible, practical and consistent. However, some languages were first specified and them implemented (ALGOL 68) or vice-versa (the already mentioned PHP). The first practice was abandoned precisely because of the problems that arise when there is no implementation to check if the specification is doable and practical.

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. For example, the LLVM [2] is an optimizing compiler infrastructure used by Apple, Google and Sony, among other big companies, and will be the target of this work.

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 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. 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 (IR). 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

One instance[2] of a bug of this type was due to how pointers work with aliasing and the resulting optimizations. In this particular case, the different semantics of UB in different parts of LLVM was causing wrong analyses of the program to be made, which resulted in wrong optimizations. This particular bug had an impact in the industry and was making the Android operating system miscompile.

Another occurrence with real consequences happened in the Google Native Client project[3] and was related to how, in the C/C++ programming languages, a logical shift instruction has UB if the number of shifts is equal to or bigger than the number of bits of its operand. In particular, a simple refactoring of the code introduced a shift by 32, which introduced UB in the program, meaning that the compiler could use the most convenient value for that particular result. As is common in C compilers, the compiler chose to simply not emit the code to represent the instruction that produced the UB.

There are more examples of how the semantics used to represent UB in today's compilers are flawed such as [3] and [4], and that is why the work we develop in this thesis is of extreme importance.

1.2 Contributions

The current UB semantics diverge between different parts of LLVM and are sometimes contradicting with each other. We have implemented part of the semantics that was proposed in the PLDI'17 paper [1] that eliminate one form of UB and extend the use of another. This new semantics will be the focus of this thesis, in which we will describe it, and the benefits and flaws it has. We will also explain how we implemented some of it. This implementation consisted in introducing a new type of structure to the LLVM IR – the **Explicitly Packed Struct** – changing the way bit fields are represented internally in the LLVM compiler. After the implementation, we measured and evaluated the performance of the compiler with the changes, which was then compared to the implementation with the current semantics of the LLVM compiler.

1.3 Structure

The remainder of this document is organized as follows: Section 2 formalizes basic compiler concepts and the work already published related to this topic. This includes how different recent compilers deal with UB, as well as the current state of the LLVM compiler when it comes to dealing with UB. Section 3 presents the new semantics. In Section 4 we describe how we implement the solution in the LLVM context. In Section 5 we present the evaluation metrics, experimental settings and the results of our work. Finally, Section 6 offers some conclusions and what can be done in the future to complement the work that was done and presented here.

[2] https://llvm.org/PR36228
[3] https://bugs.chromium.org/p/nativeclient/issues/detail?id=245

2

Related Work

Contents

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 [5, 6], that hold more specific information (such as Control-Flow Graph construction [7, 8]), 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 (the most popular front-end for the LLVM is Clang[1], which supports the C, C++ and Objective-C programming languages), 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 Intermediate Representations, 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], which resembles assembly code and is where most of the target-independent optimizations are done; the SelectionDAG[3], a directed acyclic graph representation of the program that provides support for instruction selection and scheduling and where some peephole optimizations are done; and the Machine-IR[4], that contains machine instructions and where target-specific optimizations are made.

One popular form of IR is the Static Single Assignment form (SSA) [9]. In the languages that are in SSA form, each variable can only be assigned once, which enables efficient implementations of sparse static analyses. SSA is used for most production compilers of imperative languages nowadays and, in fact, the LLVM IR is in SSA form. Since each variable cannot be assigned more than once, the IR often creates different versions of the same variable, depending on the basic blocks they were assigned in (a basic block is a sequence of instructions with a single entry and a single exit). Therefore, there is no way to know to which version of the variable `x` we are referencing to when we refer to the value `x`. The ϕ-node

[1] https://clang.llvm.org/
[2] https://llvm.org/docs/LangRef.html
[3] https://llvm.org/docs/CodeGenerator.html#introduction-to-selectiondags
[4] https://llvm.org/docs/MIRLangRef.html

solves this issue by taking into account the previous basic blocks in the control-flow and choosing the value of the variable accordingly. ϕ-nodes are placed at the beginning of basic blocks that need to know the variable values and are located where control-flow merges. Each ϕ-node takes a list of `(v, l)` pairs and chooses the value `v` if the previous block had the associated label `l`.

The code below represents a C program on the left and the corresponding LLVM IR translation of the right. It can be observed how a ϕ-node, `phi` instruction in LLVM IR, works:

```
int a;
if(c)
  a = 0;
else
  a = 1;
return a;
```

```
entry:
    br %c, %ctrue, %cfalse

ctrue:
    br %cont

cfalse:
    br %cont

cont:
    %a = phi [0, %ctrue], [1, %cfalse]
    ret i32 %a
```

This simple C program simply returns `a`. The value of `a`, however, is determined by the control-flow.

There have been multiple proposals to extend SSA, such as the Static Single Information (SSI) [10], which in addition to ϕ-nodes also has σ-nodes at the end of each basic block indicating where each variable's value goes to, and Gated Single Assignment (GSA) [11, 12], which replaces ϕ-nodes with other functions that represent loops and conditional branches. Another variant is the memory SSA form, that tries to provide an SSA-based form for memory operations, enabling the identification of redundant loads and easing the reorganization of memory-related code.

Recently, Horn clauses have been proposed as an IR for compilers, as an alternative to SSA, since despite leading to duplicated analysis efforts, they solve most problems associated with SSA: path obliviousness, forward bias, name management, etc. [13].

Optimizing compilers need an IR that facilitates transformations and offers efficient and precise static analyses (analyses of the program without actually executing the program). 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 IRs 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. A program that has UB is not a wrong program, it simply does not

specify the behaviors of each and every instruction in it for a certain state of the program, meaning that the compiler can assume any defined behavior in those cases. Consider the following examples:

```
a) y = x/0;
```

```
b) y = x >> 32;
```

A division by 0 (`a`) and a shift of an 32-bit integer value by 32 (`b`) are UB in C which means that whether or not the value of `y` is used in the remainder of the program, the compiler may not generate the code for these instructions.

As was said before, the presence of UB facilitates optimizations, although some IR's have been designed to minimize or eliminate it. The presence of UB in programming languages also sometimes lessens the amount of instructions of the program when it is lowered into assembly, because, as was seen in the previous example, in the case where an instruction results in UB, compilers sometimes choose to not produce the machine code for that instruction.

The C/C++ programming languages, for example, have multiple operations that can result in UB, ranging from simple local operations (overflowing signed integer arithmetic) to global program behaviors (race conditions and violations of type-based aliasing rules) [1]. This is due to the fact that the C programming language was created to be faster and more efficient than others at the time of its establishment. This means that an implementation of C does not need to handle UB by implementing complex static checks or complex dynamic checks that might slow down compilation or execution, respectively. According to the language design principles, a program implementation in C "should always trust the programmer" [14, 15].

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. Examples are: dividing by zero or dereferencing an invalid pointer. 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. Examples are: overflowing a signed integer or reading from an uninitialized memory position. 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 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.

One last concept that is important to discuss and is relevant to this thesis is the concept of ABI,

or Application Binary Interface. The ABI is an interface between two binary program modules and has information about the processor instruction set and defines how data structures or computational routines are accessed in machine code. The ABI also covers the details of sizes, layouts and alignments of basic data types. The ABI differs from architecture to architecture and even differs between Operating Systems. This work will focus on the x86 architecture and the Linux Operating System.

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. The presence of UB in compilers is important to reflect the semantics of programming languages where UB is a common occurrence, such as C/C++. Furthermore, it helps avoiding the constraining of the IR to the point where some optimizations become illegal, and it is also important to model memory stores, dereferencing pointers, and other inherently unsafe low-level operations.

2.2.1 LLVM

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

Additionally, 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, where each use can: evaluate to a different value, as in LLVM and Microsoft Phoenix; or return the same value, in compilers/representations such as the Microsoft Visual C++ compiler, the Intel C/C++ Compiler and the Firm representation [16].

There are some benefits and drawbacks of having `undef` being able to yield a different result each time. Consider the following instruction:

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

which in CPU architectures where a multiplication is more expensive than an addition can be optimized to:

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

Despite being algebraically equivalent, there are some cases when the transformation is not legal. Consider that `%x` is `undef`. In this case, before the optimization `%y` can be any even number, whereas in the optimized version `%y` can be any number, due to the property of `undef` being able to assume a

different value each time it is used, rendering the optimization invalid (and this is true for every other algebraically equivalent transformation that duplicates SSA variables). However, there are also some benefits. Being able to take a different value each time means that there is no need to save it in a register since we do not need to save the value of each use of `undef`, therefore reducing the amount of registers used (less register pressure). It also allows optimizations to assume that `undef` can hold any value that is convenient for a particular transformation.

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 [8, 17], 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.

Despite the need to have both `poison` and `undef` to perform different optimizations, as illustrated in Section 2.3.1, 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 (some optimizations are inconsistent with the documented semantics and with each other). This topic will be discussed later in Section 2.3.

To be able to check if the optimizations resulting from the sometimes contradicting semantics of UB are correct, a new tool, called Alive, was presented in [18]. Alive is based on the semantics of the LLVM IR and its main goal is to develop LLVM optimizations and to automatically either prove them correct or else generate counter-examples. To explain how an optimization is correct, or legal, we need to first introduce the concept of domain of an operation: the set of values of input for which the operation is defined. An optimization is correct/legal if the domain of the source operation (original operation present in the source code) is smaller than, or equal to the domain of the target operation (operation that we want to get to by optimizing the source operation). This means that the target operation needs to at least be defined for the set of values for which the source operation is defined.

2.2.2 CompCert

CompCert, introduced in [19], is a formally verified (which in the case of CompCert means the compiler guarantees that the safety properties written for the source code hold for the compiled code), realistic compiler (a compiler that realistically could be used in the context of production of critical software), developed using the Coq proof assistant [20]. CompCert holds proof of semantic preservation, meaning that the generated machine code behaves as specified by the semantics of the source program. Having a fully verified compiler means that we have end-to-end verification of a complete compilation chain, which becomes hard due to the presence of Undefined Behavior in the source code and in the IR, and due to the liberties compilers often take when optimizing instructions that result in UB. CompCert, however, focuses on a deterministic language and in a deterministic execution environment, meaning that

changes in program behaviors are due to different inputs and not because of internal choices.

Despite CompCert being a compiler of a large subset of the C language (an inherently unsafe language), this subset language, Clight [21], is deterministic and specifies a number of undefined and unspecified behaviors present in the C standard. There is also an extension to CompCert to formalize an SSA-based IR [22], which will not be discussed in this report.

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, since this is a verified compiler, 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. CompCert guarantees that the behavior of the compiled code will be exactly the same of the source code, assuming there is no UB in the source code.

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, specified the majority of the sources of UB in C, and the ones that Clight did not specify, like an integer division by zero or an access to an array out of bounds, are serious errors that can have devastating side-effects for the system and should be immediate UB anyway. If there existed the need to have deferred UB, like in LLVM, fully verifying a compiler would take a much larger amount of work since, as mentioned in the beginning of this section, compilers take some liberties when optimizing UB sources.

2.2.3 Vellvm

The Vellvm (verified LLVM) introduced in [23] 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-

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

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.

In the paper [23], the authors start by giving the LLVM IR a small-step, non-deterministic semantics - LLVM$_{ND}$ (small-step semantics refer to semantics that evaluate expressions one computation step at a time). The authors justified their choices by explaining that the non-deterministic behaviors that come with the `undef` value correspond to choices that would be resolved by running the program with a concrete memory implementation.

Although these non-deterministic semantics are useful for reasoning about the validity of LLVM program transformations, Vellvm introduced a refinement of LLVM$_{ND}$, a deterministic, small-step semantics - LLVM$_{D}$. These semantics provide the basis for testing programs with a concrete memory implementation, and give `undef` a more deterministic behavior. In LLVM$_{D}$, `undef` is treated as a `0` and reading uninitialized memory returns a `0`. It is over these LLVM$_{D}$ semantics that the Vellvm works and verifies the LLVM compiler. Since the semantics that the Vellvm uses are stricter than the LLVM ones, if a program is correct under the Vellvm semantics, than it is also correct under the LLVM semantics, but the opposite may not be true.

These semantics produce unrealistic behaviors when compiling C-style programming languages to the LLVM IR but the cases in which the semantics actually differs already correspond to unsafe programs.

2.2.4 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 [24] 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 [24]. 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.

In summary, write-write races always result in immediate UB, non atomic read-write races end with the result of the read being `undef`, and reads from a location before any write to that same location also return the value `undef`.

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

When people realized that there was a need to have UB represented in the IR, the `undef` value was created. However, having UB represented by only the `undef` is not enough since some optimizations become illegal.

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. Aside from these types of optimizations there are others that become illegal with just the `undef` value representing every type of deferred UB. 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. 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.

Another example of optimizations that are not legal with only `undef` values are hoisting instructions past control-flow. Consider this example:

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

which can be transformed to

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

Since `1/k` is loop invariant one possible optimization is hoisting the expression out of the loop. This operation should be safe since the condition in the if-statement is already checking if `k` equals 0, therefore making it impossible to have a division by 0. However, if we consider `k` to be `undef`, it is possible that the

condition `k != 0` is cleared and still have a division by 0 due to the property of `undef` having a different value each time it is used.

To be able to perform these optimizations (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

One example of the incompatibilities of the semantics of UB becomes apparent when we consider the case of Loop Unswitching and Global Value Vumbering, and the contradictions they cause on how the decisions about branches should be made.

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) [25] 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.

If `select` on a `poison` condition is Undefined Behavior, it becomes hard for the compiler to replace arithmetic operations with `select` instructions, such as in the next example, which has to be valid for any constant `C` less than 0:

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

to:

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

But if `select` on a `poison` condition is UB, then the transformed program would execute UB if `%a` was `poison`, while the original would not. However, as we have seen previously, `select` and branch on `poison` not being UB makes GVN optimizations illegal, which would make the transformation above unsound. It is also sometimes desirable to perform the inverse transformation, replacing `select` with arithmetic instructions, which makes it necessary for the return value of `select` to be `poison` if any of the arguments is `poison`, contradicting the branch to `select` transformation. Currently, different parts of LLVM implement different semantics for this instruction, originating end-to-end miscompilations.

Aside from the problems related to the semantics of `poison`, there are other instances where the simultaneous presence of both `undef` and `poison` in the same instruction result in wrong optimizations. For example, LLVM currently performs the following substitution:

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

to:

```
%v = %x
```

However, this optimization is wrong because `%x` could be `poison`, and `poison` is a stronger form of deferred UB than `undef`, meaning that we might be introducing stronger UB when there is no need to.

In Table 2.1 we can observe the different alternative of semantics for `select`. The table assumes that branch on `poison` is UB. As we can see, there are no columns where all the rows are checked, meaning that there is no single semantics in which all the transformations we are interested in doing are correct. As can be concluded, something else needs to be implemented in the IR to solve this problem.

Table 2.1: Different alternative of semantics for select[6].

	UB if `%c poison` + conditional `poison`	UB if `%c poison` + `poison` if either `%x`/`%y poison`	Conditional `poison` + non-det choice if `%c poison`	Conditional `poison` + `poison` if `%c poison`	`Poison` if any of `%c`/`%x`/`%y` are `poison`
SimplifyCFG	✓		✓	✓	
Select → control-flow	✓	✓			
Select → arithmetic		✓			✓
Select removal	✓	✓		✓	✓
Select hoist	✓	✓	✓		
Easy movement			✓	✓	✓

2.3.4 Bit Fields and Load Widening

Another problem `poison` creates is when 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

[6] http://lists.llvm.org/pipermail/llvm-dev/2017-May/113272.html

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`.

2.4 Summary

As we can see, UB, which was implemented in the LLVM's IR to enable certain useful optimizations, currently has inconsistent semantics between different LLVM parts and between both forms of deferred UB. Aside from that, with the current implementation, some optimizations require that UB have specific semantics that make some other optimization become unsound. All this has led to conflicting assumptions by compiler developers and to end-to-end miscompilations. The rest of this work consists in our attempt to solve these problems.

3

LLVM's New Undefined Behavior Semantics

Contents

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 to change the semantics of LLVM to a new semantics that discards `undef` and only uses `poison` instead. To be able to solve some problems that come from the removal of `undef`, a new instruction `freeze` is introduced 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`, and branching on `poison` is immediate UB. The introduction of the `freeze` instruction which was already created as patches[1][2][3] 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` (since the optimizations are done in passes, if one optimization uses `freeze` to remove `poison` from the program, consequent optimizations cannot take advantage of it).

3.1 Semantics

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

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

[1] https://reviews.llvm.org/D29011
[2] https://reviews.llvm.org/D29014
[3] https://reviews.llvm.org/D29013

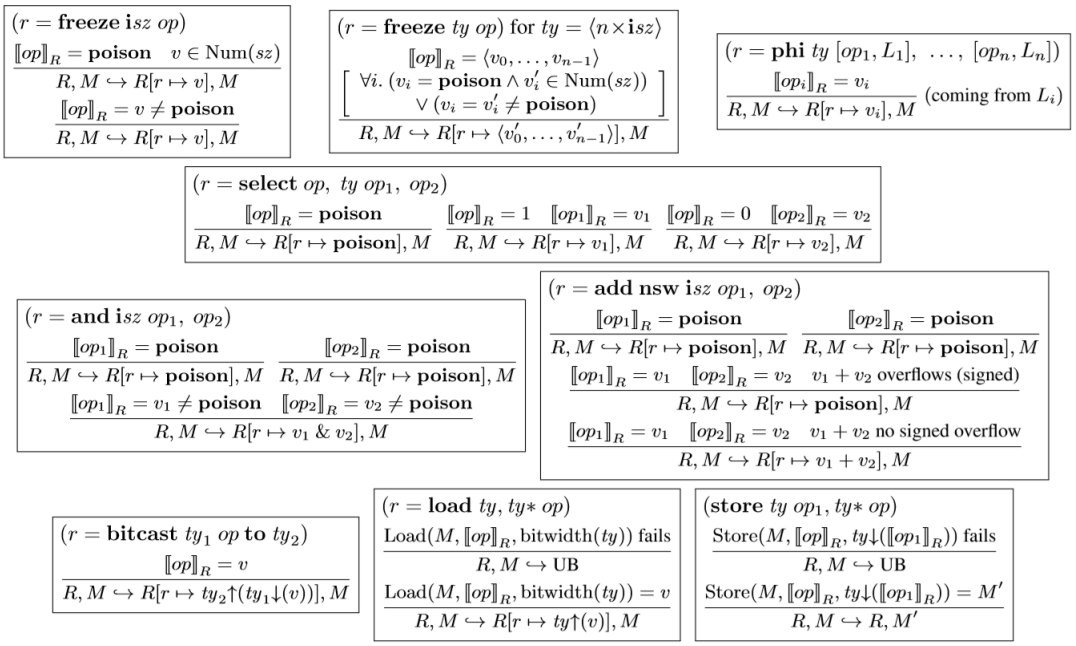

Figure 3.1: Semantics of selected instructions [1].

Here, Num(sz) refers to any value between 0 and 2^{sz}, where sz refers to the bit width of the value. **[**isz**]** 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 3.1, 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 **[poison]**$_R$ = **poison**, for `poison`. The `Load`(M, p, sz) operation only returns the loaded bit representation if p is a non-poisonous, previously allocated pointer to a valid block of at least sz bits wide in memory M. The `Store`(M, p, v) operations only stores the value v at the position p in memory M and returns the updated memory, if p is a non-poisonous, previously allocated pointer to a valid block of at least **bitwidth(**v**)** bits wide.

Consider the `sdiv` instruction with the `exact` flag, which states that the result of the division is `poison`

if the remainder is different than `0`:

$$(r = \textbf{sdiv exact}\ ty\ op_1, op_2)$$

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

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

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

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

This example shows the semantics of the `sdiv` instruction with the `exact` keyword and how it affects Reg and Mem on x86 processors, in the case of deferred Undefined Behavior. Here we can observe that if either op_1 or op_2 are `poison`, then the result is `poison`. The `exact` keyword makes the result of the instruction `poison` if its remainder is different from 0 (as was said before). There are two more cases that are not represented in which the instruction is not defined: the case where op_2 is 0, since division by 0 is immediate UB; and the case where op_1 is equal to **MIN_INT** and op_2 is equal to -1, since in this case the result would overflow, which in x86 processors also results in immediate UB, causing a crash.

In Figure 3.1, there can also be seen two meta operations, used to define the semantics of instructions: $ty\downarrow$ (function where its domain is the set of values of type **[*ty*]** and its codomain is the set of bit vectors representation which represent those values) and $ty\uparrow$ (function where its domain is the set of bit vectors of **bitwidth(**ty**)** size and its codomain is the set of values of type **[*ty*]**):

$$\begin{aligned} ty\downarrow &\in [\![ty]\!] \rightarrow [\![\langle \textbf{bitwidth(}ty\textbf{)} \times \mathbf{i}1 \rangle]\!] \\ ty\uparrow &\in [\![\langle \textbf{bitwidth(}ty\textbf{)} \times \mathbf{i}1 \rangle]\!] \rightarrow [\![ty]\!] \end{aligned}$$

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

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

$$\mathbf{i}sz\uparrow(b) \text{ or } ty*\uparrow(b) = \begin{cases} \textbf{poison} & \textbf{if } \exists i.b[i] = \textbf{poison} \\ (std) & \text{otherwise} \end{cases}$$

$$\langle sz \times ty \rangle\uparrow(b) = \langle ty\uparrow(b_0) \ \ldots \ ty\uparrow(b_{sz-1}) \rangle$$

$$\text{where } b = b_0 +\!\!+ \ldots +\!\!+ b_{sz-1}$$

These two meta operations were defined in [1]. $ty\downarrow$ transforms poisonous base types into a bitvector of all `poison` bits (λ means every bit), and into their standard low-level representation, otherwise. In the case of vector types, $ty\downarrow$ transforms values element wise, where $+\!\!+$ denotes the bitvector concatenation. 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, $ty\uparrow$ works like $ty\downarrow$, transforming the values element-wise.

3.2 Illustrating the New Semantics

In Section 2.3, we discussed the benefits but also the conflicting optimizations brought by the introduction of `poison` values to the LLVM IR. 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.

3.2.1 Loop Unswitching and GVN

It was previously showed in Section 2.3 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. Take a look at the previous example of Loop Unswitching, but with a `freeze` instruction on the value of the conditional branch:

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

To put it differently, if `c2` is a defined value then the optimization is correct. If `c2` is a `poison` value, then the `freeze` instruction would "freeze" `c2` and non deterministically choose an arbitrary value of the same type of `c2`, therefore transforming the condition of the jump into a defined value. Since the original code was refined and the UB was taken away, the optimization is now correct.

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 the introduction of a `freeze` instruction in the following way:

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

is transformed into:

```
   %c2 = freeze %c
   br %c2, %true, %false

true:
   br %merge

false:
    br %merge

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

Although it was decided that `select` conditionally returns `poison` by selecting either value `%a` or `%b` according to the condition `%c`, in the case where the condition `%c` is itself `poison`, `select` triggers immediate UB. This is because of a design choice to define branch on `poison` to be immediate UB, as was discussed in the beginning of Section 3 . By introducing a `freeze` instruction in the transformation, therefore "freezing" the condition, we can take away the UB that could stem from it, consequently refining the semantics of the program, making the optimization legal.

3.2.3 Bit Fields

As was addressed before, in Section 2.3.4, 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. However, if we access bit fields the way they are in the LLVM IR we might "poison" adjacent bit fields. Right now the LLVM IR loads or stores the entire word where a certain bit field resides. If we have a structure with two 16-bit bit fields, both bit fields will be stored in a single 32-bit word. To access the field stored in the 16 least significant bits, the generated code will be the following:

```
x = mystruct.myfield;
```

into

```
%val = load i32, i32* %mystruct
%val2 = and i32 %val, 65535
%myfield = trunc %val2 to i16
store %myfield, %x
```

By loading the 32-bit word as a whole and if one of those bit fields is `poison`, the other bit field is already contaminated.

The proposal of [1] was to simply "freeze" the loaded bit field value before combining it with the new one, like the following example:

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

into

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

However, when we load the entire word where the bit field that needs to be accessed is, all the adjacent bit fields are already being contaminated by `poison` even before `freezing` the word. This means that we would first `poison` the entire word, then by `freezing` it we would be choosing a non-deterministic value for every bit, altering the value of every bit field in that word, which is what we were trying to avoid in the first place.

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 the previous structure `s`, defined in Section 2.3.4 would be:

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

while the new structure would be represented by:

```
%struct.s = type { i3, i4, i1, [3 x i8] }
```

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. This padding is automatically appended by the ABI, referred in Section 2.1. The ABI covers the sizes of data types and, in this case, it determines the amount and size of words that are needed to store the bit fields of a structure and inserts the needed padding at the end.

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. 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`.

There are other ways of lowering bit fields to the LLVM IR, such as using vectors or the structure type, but these are currently not well supported by the LLVM's backend [1]. There is no risk of "poisoning" the adjacent bit fields with these solutions - by loading the fields as a vector (for example) one element does not contaminate the others. In this next example we can see a `store` using a vector of 32 bits for the `load` (loading each field separated from the others), instead of loading and storing a single 32-bit word, as is usual:

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

The `insertelement` instruction inserts an element into the aggregate type. However, the example is summarized since `insertelement` can only insert one element to an aggregate type at a time. So, in the case of a bit field with a size of multiple bits, the same amount of `insertelement` instructions would need to be emitted (bit field of 5 bits implies 5 `insertelement` instructions).

3.2.4 Load Combining and Widening

Also in Section 2.3.4 we 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
```

The `extractelement` instruction is the opposite of the `insertelement`, extracting one element of an aggregate type at a time.

3.3 Cautions to have with the new Semantics

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.

The transformation of:

```
x = a << b;
y = freeze(x);
while(...) {
  call(y)
}
```

into:

```
while(...) {
  x = a << b;
  y = freeze(x);
  call(y)
}
```

is not allowed, for example.

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. Consider the next piece of code:

```
while(c)
  x = 1/y;
```

In this example, since `1/y` is loop invariant, we want to hoist it out of the loop. To do this we run a static analysis over `y` to make sure that we will not ever be executing a division by `0`. If `y` turns out to be `poison` though, the static analysis will consider the transformation safe, but we would be introducing UB.

In conclusion, there is a trade-off in the semantics of static analyses. We do not want to take `poison` values into account to not always return the worst value possible but, on the other hand, by not taking them into account we cannot depend on the results of the analyses since they can be wrong.

4

Implementation

Contents

This section details how the proposed solution was implemented into the LLVM compiler, starting by explaining what needs to be changed inside the LLVM to implement a new feature (Section 4.1). We started by implementing the solution to `poison` in the bit fields. That task, however, proved to be complex enough to be in a document of its own. With this in mind, this section will focus on the two proposed solutions to deal with `poison` in bit fields: 4.2 explains the attempt at a Vector Loading solution and talks about the pros and cons of this approach, and 4.3 explains in detail the new type of a structure we introduced in LLVM and how it solves `poison` at the LLVM IR level. Since the LLVM compiler is mostly written in C++ that was the language we used.

Our implementation can be found at `https://github.com/FilipeAz/llvm-project`.

4.1 Internal Organization of the LLVM Compiler

The source code of the LLVM compiler is incredibly complex, with tens of thousands of files, often with thousands of lines of code each. To change or implement a new feature into the compiler, a great number of parts of the compiler need to be updated to accommodate the new changes. As Section 2.1 describes, the LLVM compiler is divided into 3 different layers. These segments are the front-end, which in our case the focus will be on the C language family front-end, Clang, the middle-end and the back-end, our target being the x86 architecture. Since we wanted to update/create a type in the IR, our feature affected all 3 layers of the compiler.

We can divide the changes to the Clang front-end into two. First of all, the parser has to be updated to generate the extra information. In our case, since what we are is changing the structure type, we have to make sure that every structure that contains bit fields is flagged accordingly. Secondly, the implementation of what the new IR should look like: this means updating the emission of instructions to the IR and, in our case, also updating and making sure that the information on structures and sizes of its fields that is present in the current IR implementation is not lost.

The middle-end layer mostly has to do with optimizations to the LLVM IR. When it comes to the architecture-independent optimizations, changing how a type is represented, introducing a new type or even introducing a new instruction, such as the `freeze` instruction, to the LLVM IR implies that a great deal of optimizations also need to change to recognize it.

Finally, the back-end deals with the emission of the SelectionDAG, the emission of assembly code and architecture-specific optimizations. The SelectionDAG is the next IR in the LLVM pipeline (an intermediate representation where each instruction is represented in a data dependence DAG). SelectionDAG nodes usually translate directly from LLVM IR instructions, but when implementing a new type or a new instruction in the IR, the emission of the corresponding nodes need to be updated/implemented in the SelectionDAG as well. The emission of assembly code follows the rules of emitting SelectionDAG

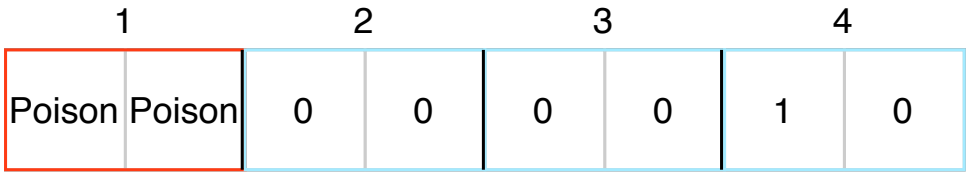

Figure 4.1: An 8-bit word being loaded as a Bit Vector of 4 elements with size 2.

nodes when it comes to a new type/instruction, and the same is true for the architecture-dependent optimizations.

4.2 The Vector Loading Solution

As was stated before, the first solution to prevent the propagation of `poison` between bit fields of the same word was to `freeze` the loaded word. This doesn't work since when we load the word in the first place, we are already spreading `poison` to the other bit fields. Having realized this we quickly turned to the second solution presented by [1], Vector Loading. Vector Loading is in theory a better option than "freezing" the loaded word since there is no need to introduce an extra instruction, `freeze`, and also provides load widening optimizations.

Vector Loading means we first need to transform the word we want to load in a bit vector, where every position of the vector corresponds to a bit of the same word. Then the word could be loaded as a vector, and every bit of the same bit field could be rebuilt by `oring` the corresponding bits, eliminating the problem of `poison` contamination and the need to `freeze` the load. Unfortunately, in the LLVM IR the load of a variable and the variable have to be of the same type. With this in mind we come to the conclusion that every word of the structure that contains a bit field needs to be declared as a vector of bits.

We started by noticing that it didn't necessarily have to be a bit vector, but simply a vector where every element had the size of the greatest common divisor of every bit field in that word. For example, if there were 2 bit fields in an 8 bit word, one with a size of 2 bits, and another with a size of 6 bits, the vector to load could be a vector with 4 elements, where each element was of size 2, meaning that the first bit field was stored in the first element and the second was divided between the second through fourth elements. Figure 4.1 provides a visual representation of this example, where the size 2 bit field is `poison` and the size 6 bit field (divided between the second through fourth elements of the vector) is storing the value `2`. This way the amount of work needed to build the bit field from their respective bits would be reduced. Having a large greatest common divisor between every bit field in the same word was not common but it was also not impossible.

Despite this, and as we witnessed before, there is sometimes **padding** at the end of the words, after

the bit fields. This padding exists because the size of words is given by the ABI (mentioned in Section 3.2.3), which decides that every word is a multiple of 8 bits, to be able to more easily translate the IR to machine code. In fact, at the processor level the size of words is even more restricted, with the x86 architecture needing the size of a word to be a power of 2 starting at 8 bits, or a byte. A simple example is the following C code:

```
struct {
  int i : 15;
  int j : 15;
}
```

As can be seen, both bit fields would fit in a 30-bit sized word. 30 is not a multiple of 8, so the ABI decides that a 32-bit word is necessary with 2 bit of padding at the end.

On top of this restriction, the ABI decides the size of the word where each field will be stored depending on the type of that same field and depending on the structure being packed or not (which we will be talking about later). So the following structure written in C:

```
struct {
  int i : 3;
  char c : 6;
}
```

Will produce the following LLVM IR:

```
type { i8, i8, [2 x i8] }
```

This means that even though a 16-bit integer would have sufficed to store both bit fields, where the char field would have an offset of 3 bits, the ABI decides to separate them in two different 8-bit words (the vector at the end is padding also appended by the ABI to bring the size of the structure to 32 bits since in some architectures unaligned accesses to memory are not allowed, making the padding at the end obligatory). This happens because a char's size is usually 8 bits, and here we would have a char technically occupying more than one 8-bit word by being stored in the last 5 bits of the first 8-bit word and in the first bit of the second 8 bit word. Of course, in this particular example of how the ABI chooses the sizes of the words there would be no problem in accessing the bit fields the standard way, since both bit fields are stored in different words. So let's take a look at a structure where the padding conflicted with our greatest common divisor idea:

```
struct {
  int i : 3;
  int c : 6;
}
```

With the corresponding LLVM IR:

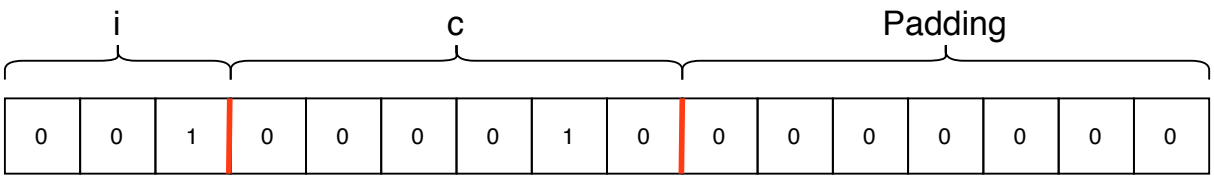

Figure 4.2: Padding represented in a 16-bit word, alongside 2 bit fields.

```
type { i16, [2 x i8] }
```

In this case, since both bit fields are of the integer type, and neither occupy more than one 32-bit word, the ABI decides to emit a single 16-bit word to store them. This means that we have the bit field *i* starting at the offset 0, the bit field *c* starting at the offset 3, and then 7 bits of padding, from the 10th bit until the 16th bit, as can be seen in Figure 4.2. We can observe that the padding is 7 bits. Even though the greatest common divisor between the size of both bit fields is 3, if we were to create a vector load to access either bit field `i` or `c`, each element of the vector had to be smaller than the desired size of 3 bits since with padding the size of the word is 16 bits and the greatest common divisor between both fields' sizes and the size of the word is just 1.

With all this in mind, we come to the conclusion that due to padding that more often than not exists at the end of words, whose size was computed by the ABI, the greatest common divisor between every bit field of a word and the padding was usually 1.

We previously mentioned that the size of each word generated by the ABI also depends on the structure being **packed** or not. In the C programming language a structure can either be packed or non packed. On top of the difference of size in emitted words, a non packed structure aligns every one of its fields to the address boundaries of their types, by adding padding after each one (if needed), making sure that every memory access is aligned, which in turn results in faster accesses. If we have a structure composed of 3 fields, a char, an int and a char, the compiler will add 3 bytes of padding to the first char so that the int field, which has a size of 4 bytes, can be stored in a multiple of 4 address, and add 3 bytes of padding to the last char so that the size of the structure can be a multiple of 4 too, effectively making the size of the structure 12 bytes instead of 6, double of what is actually needed. We can tell the compiler to not add this padding in C by specifying that the structure we are creating is packed, by passing the directive `__attribute((__packed__))__` to the header of the structure. The LLVM IR represents packed structures by adding a < and a > to the beginning and the end of structures. In the previous example, with a packed structure we would only occupy 6 bytes.

The above example, of a structure with a char and an integer bit fields where both bit fields were stored in different 8-bit words is true for a non packed structure but, in fact, the size of the word generated by the Clang ABI changes to a 16 bit word in case of a packed structure. In this case there would be a need to vector loading to not `poison` the other bit field.

Aside from the road blocks the ABI can create, the main problem that kept us from trying this approach was the lack of LLVM IR support to load or store more than one element of a vector or array in a single `load` or `store` instruction. This means that in the worst case scenario of a bit field that occupies 63 bits, the number of total instructions that would be needed to load its value was 252 extra instructions in the IR since every bit of the bit field would need a `gep` instruction to figure out the place in memory, a `load` instruction to actually load that bit, followed by a `shift` and an `or` instructions to build the correct value stored in the bit field. Of course the number of instructions in the assembly code would be the same but this implementation would clutter the IR and make it hard to optimize, while also making it very hard to read for the developers.

Another problem we haven't discussed here is the limitations regarding the access of information needed to build these vectors in the LLVM source code. Not being able to access, for example, the size of all the fields of the structure in the function where we needed to figure out the size of each element of the vector for the load would force us to alter multiple functions across different levels of the front- and the back-end of the compiler.

4.3 The Explicitly Packed Structure Solution

After being faced with all the problems explained above we decided to try the new structure type in the LLVM IR solution.

As was already discussed in Section 3.2.3, the goal of this solution was to delay 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 take the information computed by the ABI and 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; also because the IR usually does not emit the final padding at the end of the structure, which would only appear in the generated assembly, sometimes confusing programmers). This padding in between words refers, for example, to the padding of 3 bytes inserted by the ABI after a char field so that every field is stored in a multiple of 4 address, as discussed in Section 4.1. We use slashes to denote that a particular structure has explicit packing.

This approach seemed to work when it came to visible results of programs. However, the back-end was not emitting the correct code, or at least the code we wanted it to emit (the code in itself was valid). This is easily understandable if we look at an example. 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:

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

Instead of:

```
%struct.anon = type { i8, [3 x i8] }
```

For this structure with two bit fields, a char and an int, both with size 2, the generated assembly allocated 8 bits for each field. This obviously makes sense since the processor can only process things in bytes, or multiple of 8 bits. It was fairly simple to resolve this problem however, as all we had to do was to group each bit field by word according to its size, and initialize each word accordingly, before generating the information for the next IR in the pipeline. Before this new structure, the LLVM IR took this into account: as we can see by the old LLVM IR code, that 8-bit word stored the values of both the *c* and *i* bit fields and was ready to be emitted to assembly.

As an example, 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 (%struct.anon, %struct.anon* @s, i32 0, i32 1)
store i2 1, i2* %1
```

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

```
%1 = i8* getelementptr (%struct.anon, %struct.anon* @s, i32 0, i32 0)
%2 = load i8, i8* %1
%3 = and i8 %2, -13
%4 = or i8 %3, 4
store i8 %4, i8* %1
```

where `@s` 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. As can be observed by the example, what needs to happen to store a value into a bit field are several instructions in the following order: first we need to load the word where the bit field is stored (minimum of 8 bits, or a byte), then we need to check the offset of the bit field in the word, so if the word is 8 bits and the bit field, in this case with size 2, only starts on the third bit of the word, we need to replace the third and fourth bits with zero (erasing the

previous content of the bit field) with an `and` instruction. Then we need to prepare the value we want to store, and since the offset is 2, we need to shift that value by 2 (value 4 in the `or` instruction). Finally we `or` the value and the original word (with the third and fourth bits now equal to zero) and store the result in memory.

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. The next intermediate representation is the SelectionDAG. Before, the translation to SelectionDAG nodes from the IR was straight-forward, with almost a one to one translation of the IR instructions to SelectionDAG nodes. Now we have to add the bit wise operations ourselves to the SelectionDAG: basically, every time we were processing either a load or a store node we would check if the address we were accessing was a bit field and insert the different bit-wise operations nodes in the DAG, if necessary.

The main benefit of this new representation is to stop the 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 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 at 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

Evaluation

Contents

During the development of our implementation, we used the LLVM regression tests to check the correctness of the compiler. In the end, all tests passed except the ones that checked the IR of structure types that contained bit fields, in which case the only difference was that the old IR did not have the '\' and '/' characters and the words used to store the fields were now different.

After implementing our solution we used the LLVM Nightly test-suite to test it. 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.

Due to the way that we marked the explicitly packed structures in the LLVM IR (with the '\' and '/' characters) it is much easier to spot structures that contain bit fields there than it is in the C source code. With this in mind we used the gllvm[1] tool to check which tests contained bit fields. The gllvm is a tool that provides wrappers that work in two phases: first they invoke the LLVM compiler as usual, then they call the LLVM bitcode compiler that generates bitcode files of each object file and store them in a dedicated section of the object file. When object files are linked together, the contents of the dedicated sections are concatenated. After the build completes, we can read the contents of the dedicated section and link all of the bitcode into a single whole-program bitcode file.

Since the LLVM test-suite is built using cmake, it is not easy to produce LLVM IR from the source files, due to the compiler checks that cmake runs over the produced files. By using gllvm as the compiler, we run make in the LLVM test-suite and then produce bitcode files from the object files, which could then be translated to LLVM IR files using the LLVM disassembler (llvm-dis), we were able to pinpoint which were the relevant tests to run.

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[2]. 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.

In fact, the GCC benchmark helped us debug our compiler thoroughly during development. To be able to compile it we had to face and correct various bugs that we would never encounter by just running the LLVM regression tests or the LLVM test-suite. Unfortunately, pinpointing what is the problem in a program with a source code of this dimension is not easy. To help us find the part of the code that created a problem we started using the C-Reduce[3] tool. Introduced by [26] as a way to help reporting compiler bugs, where one needs to find a small case test that triggers the bug, C-Reduce is a tool that takes large C, C++ or OpenCL files that have a property of interest (our property of interest is triggering

[1] https://github.com/SRI-CSL/gllvm
[2] https://people.csail.mit.edu/smcc/projects/single-file-programs/
[3] https://embed.cs.utah.edu/creduce/

a specific compiler bug) and automatically produces a much smaller file that has the same property.

The use of the C-Reduce tool was a step in the right direction. However, with a file this large, C-Reduce would take a long time to produce a file which we could work on. And even when it finally produced that file, that program would only reproduce the first bug we found while trying to compile GCC. Fortunately, we discovered another tool, created by the same researchers that created C-Reduce, called Csmith[4]. The Csmith tool was introduced in [27] and is a tool that can generate random C programs. Since it is a random program generator, it has options to tune the properties of the generated code, like the C subset, the size of the program, and other properties. One of these other properties was to only generate programs that have bit fields.

Using a combination of Csmith and C-Reduce, we debugged the whole GCC single source file. On top of that we ran both tools a couple more times to make sure that only a very small amount of bugs persisted with the -O0 flag, in which the compiler doesn't run any optimization. After making sure that the GCC single source file compiled and that executable actually worked as the GCC compiler (comparing code produced by the GCC compiled with our compiler with code produced by the GCC compiled with the LLVM commit from which we forked our own) we started running the tests.

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, a class designed to emit poor code quickly, which is the case with the -O0 flag, where the quality of the generated code is irrelevant when weighed against the speed at which the code can be generated. The reason for this is that we now need to store some information present in the `getelementptr` instruction to emit SelectionDAG nodes that represent the instructions that access bit fields, which meant that the FastISel class had to return the job of emitting code to the SelectionDAGBuilder class. This is only needed when the `getelementptr` instruction is doing arithmetic to access bit fields but since we have no way of knowing when a field is a bit field in the original LLVM, we decided to disable FastISel for every `getelementptr`

[4] https://embed.cs.utah.edu/csmith/

instruction.

On top of that, we also disabled some optimizations that malfunction with our new implementation of structures. These optimizations are SCCP (Sparse Conditional Constant Propagation), an optimization that removes dead code and tries to propagate constants in the program, and GVN (Global Value Numbering), explained in Section 2.3.2 that from our experience generated wrong code when optimizing code with bit fields.

There are two other optimizations that generated wrong code with our new type of structure but that we unfortunately cannot disable. This is because these optimizations clean the code and put it in a canonical IR form on which subsequent optimizations depend (for ease of implementation of those optimizations), meaning that these next optimizations would also be disabled. These two optimizations are SROA (Scalar Replacement of Aggregates) and the InstCombine optimization. The SROA is an optimization that tries to identify elements of an aggregate that can be promoted them to registers, and promotes them. The InstCombine optimization simply combines instructions to form fewer, simple instructions. We subsituted SROA for Mem2Reg, another optimization similar in nature that did not generate wrong code with the new structures. InstCombine however does not have other similar optimization that could be switched out for so in this case we simply disabled it in any function that had an access to a bit field in our implementation, where in the original LLVM we left InstCombine as is. This will have some impact in the results of the -O3 flag, which will be discussed later.

Unfortunately, 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.

The results will be shown according to their change in percentages, with a negative percentage indicating that performance improved and a positive percentage indicating that performance degraded. We will also be separating the benchmarks/applications from the micro-benchmarks, since the benchmarks are more important to the evaluation.

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 $\pm 2\%$ as can be seen in Figure 5.1.

With the micro-benchmarks we decided to only represent the tests that took more than 5 seconds to compile, shown in Figure 5.2, as any small change in the rest of the tests would equate to a bigger difference that does not represent the reality. Even so, this can still be observed by the almost $+6\%$

change in the micro-benchmark “GCC-C-execute-20000815-1”.

The results with the -O3 flag were identical to the ones with -O0. The benchmarks were in the range of $\pm 1\%$ as can be seen in Figure 5.3. Again we only show the micro-benchmarks that took more than 5 seconds to compile and there are bigger changes in the smallest programs (Figure 5.4).

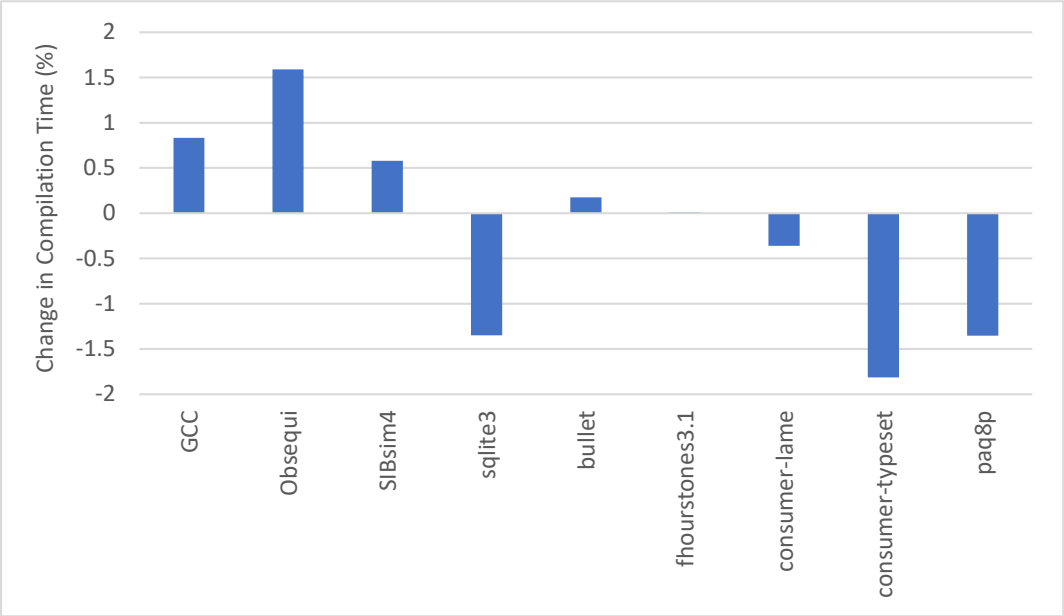

Figure 5.1: Compilation Time changes of benchmarks with -O0 flag.

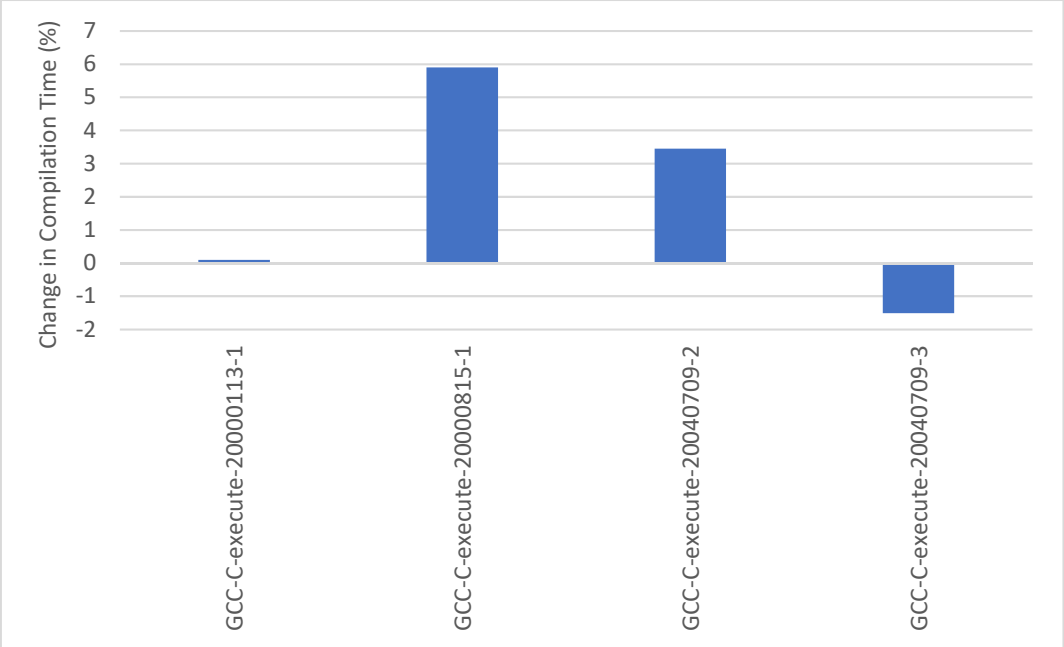

Figure 5.2: Compilation Time changes of micro-benchmarks with -O0 flag.

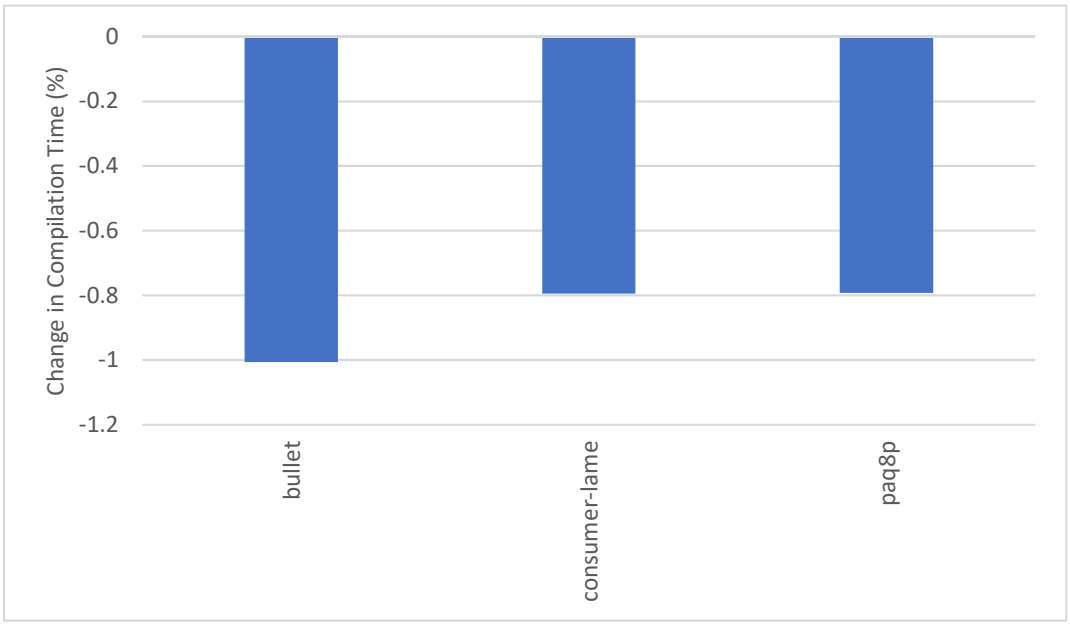

Figure 5.3: Compilation Time changes of benchmarks with -O3 flag.

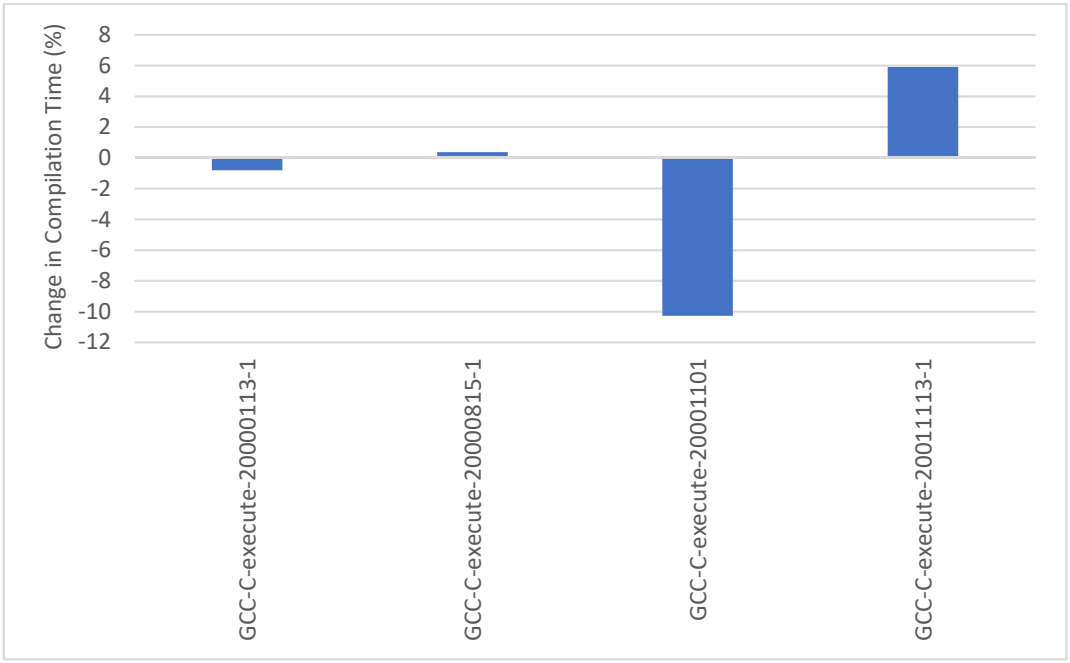

Figure 5.4: Compilation Time changes of micro-benchmarks with -O3 flag.

5.3 Memory Consumption

As was explained before, to estimate peak memory consumption we used the ps tool and recorded the RSS and VSZ columns every 0.02 seconds. The RSS (Resident Set Size) column shows how much memory is allocated to a process and is in RAM. The VSZ (Virtual Memory Size) column shows how much memory the program can access, including memory that is swapped out, memory that is allocated, and memory from shared libraries.

For all benchmarks with the -O0 flag peak memory consumption was unchanged, both RSS and VSZ, all within the $\pm 1\%$ range, as can be seen in Figures 5.5 and 5.6. The micro-benchmarks saw a RSS value fluctuating between $\pm 4\%$ (Figure 5.7) while the VSZ value maintained values in the $\pm 0.4\%$ (Figure 5.8).

Regarding the results with the -O3 flag, the peak memory consumption for the benchmarks kept a $\pm 2\%$ 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 5.9 and 5.10. This benchmark is a C++ program with only 2 bit fields and we verified that this difference in peak memory consumption remained even when we transformed the bit fields 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, either in the StructType class or the DataLayout class. 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 (Figures 5.11 and 5.12).

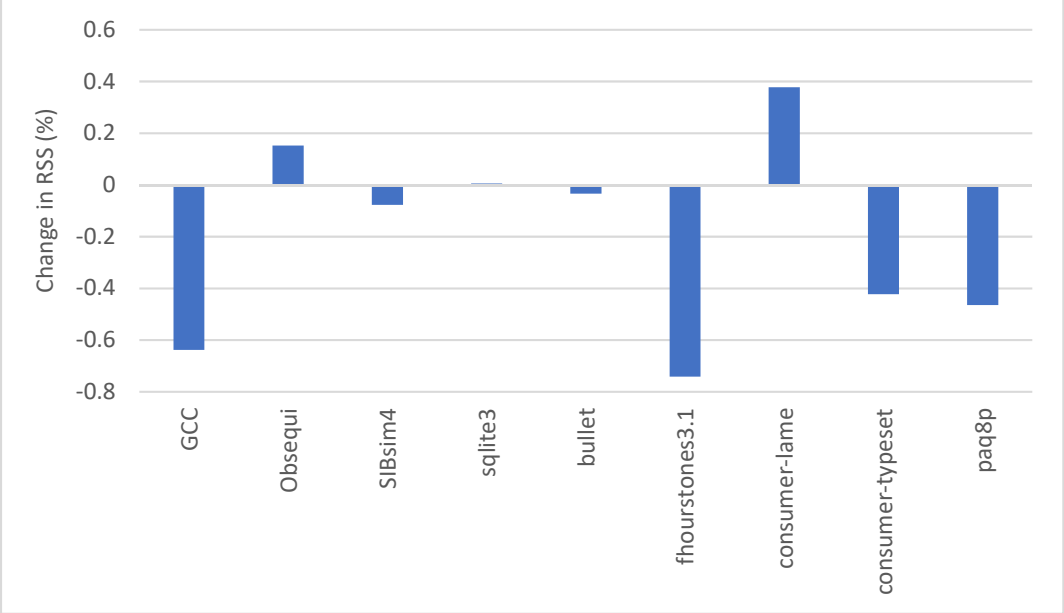

Figure 5.5: RSS value changes in benchmarks with the -O0 flag.

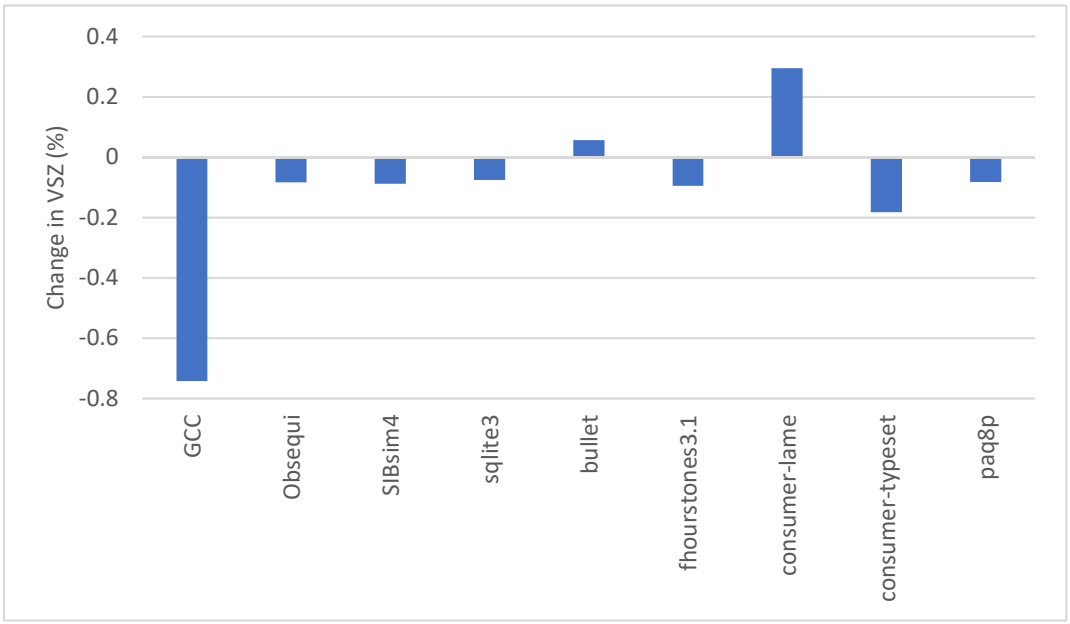

Figure 5.6: VSZ value changes in benchmarks with the -O0 flag.

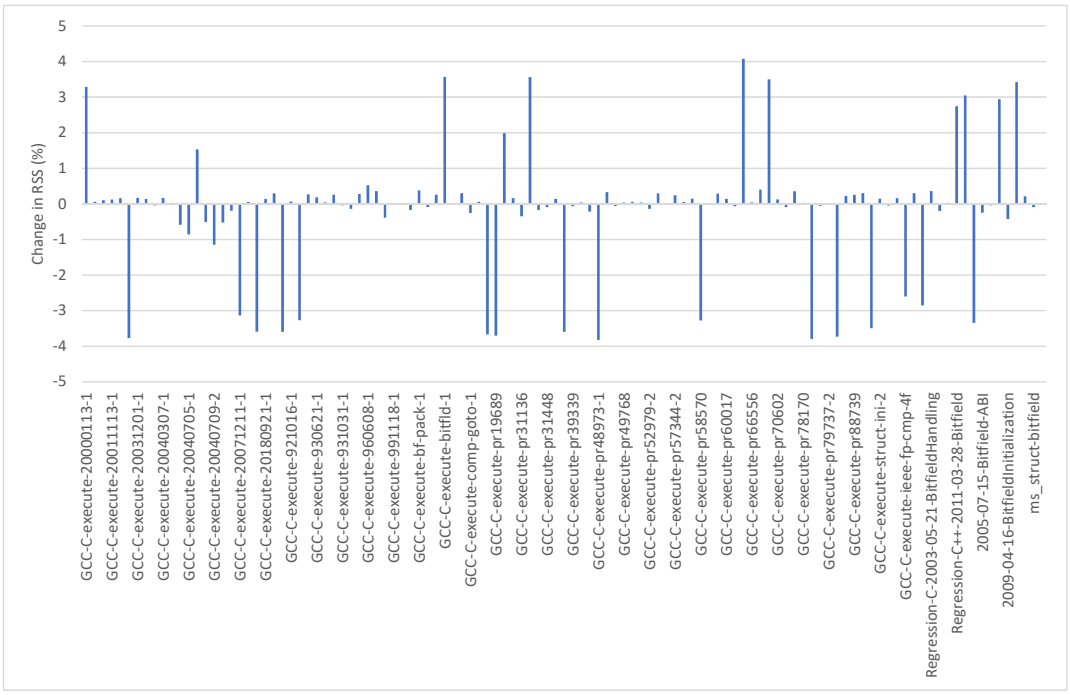

Figure 5.7: RSS value changes in micro-benchmarks with the -O0 flag.

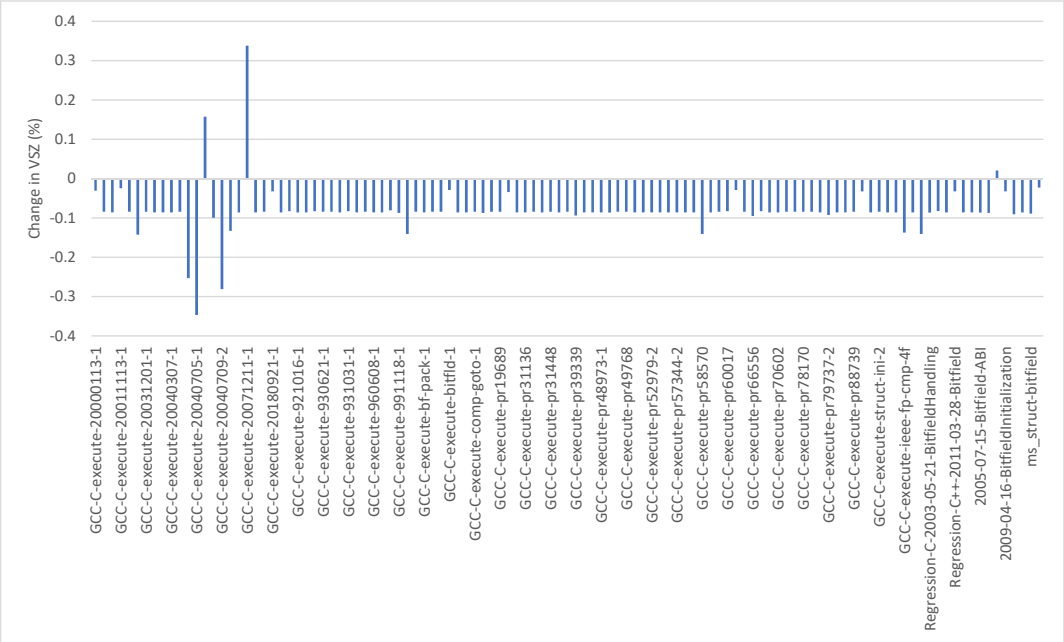

Figure 5.8: VSZ value changes in micro-benchmarks with the -O0 flag.

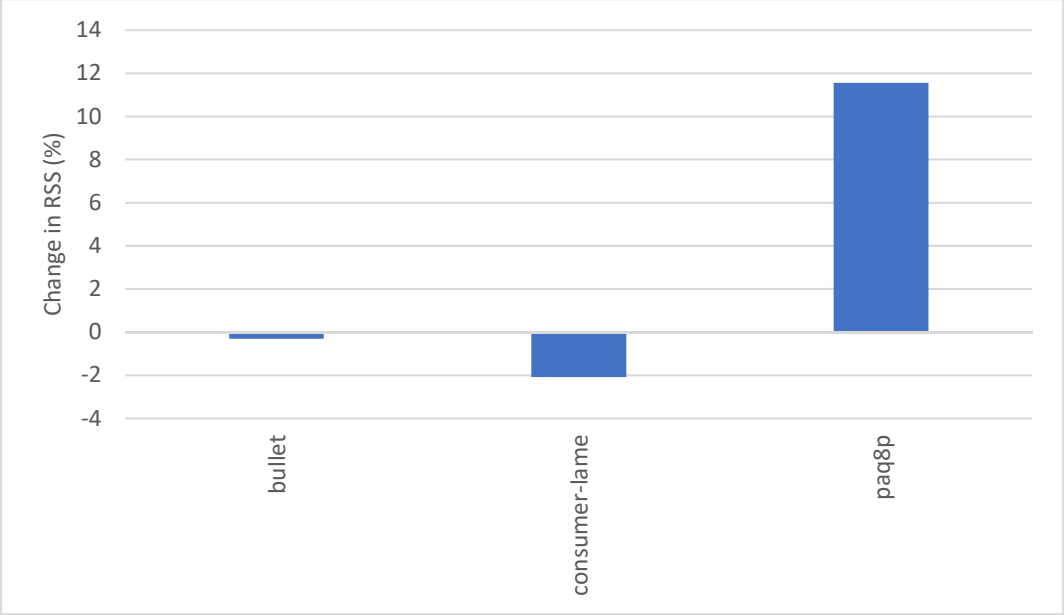

Figure 5.9: RSS value changes in benchmarks with the -O3 flag.

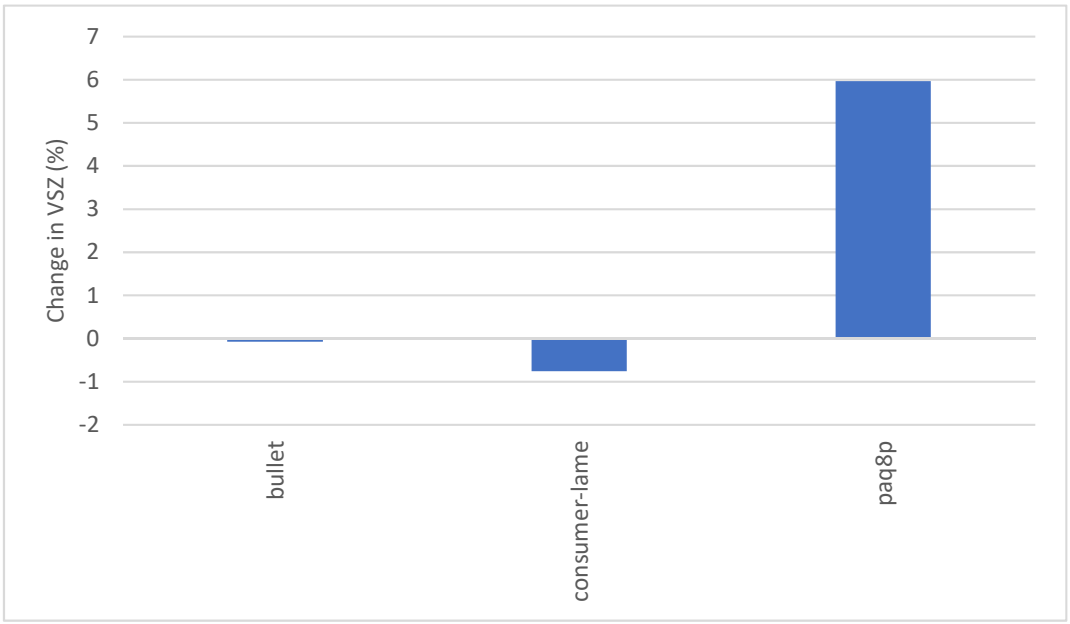

Figure 5.10: VSZ value changes in benchmarks with the -O3 flag.

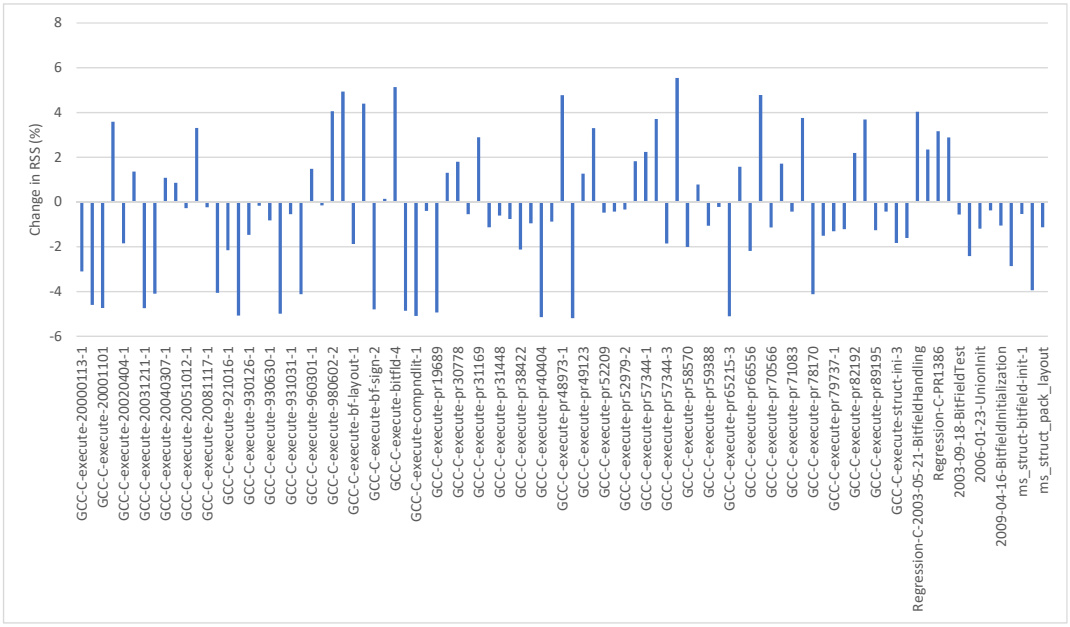

Figure 5.11: RSS value changes in micro-benchmarks with the -O3 flag.

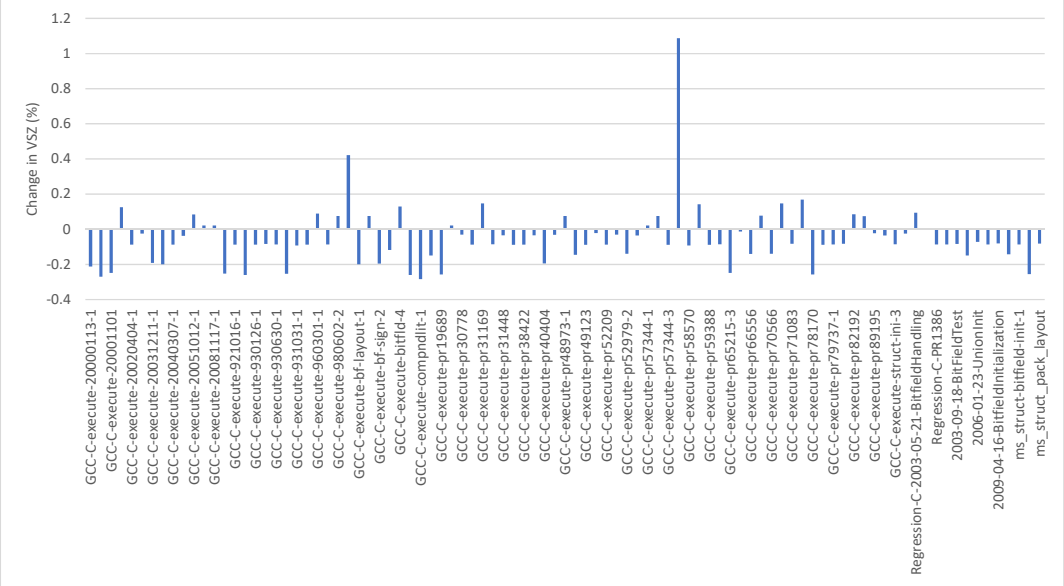

Figure 5.12: VSZ value changes in micro-benchmarks with the -O3 flag.

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 when compared to the .o files compiled with the original LLVM: 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 $\pm 1.6\%$ as can be observed in Figure 5.13, 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 in particular extensively tests accesses to fields of structures meaning that almost every function has an access to a bit field. Since InstCombine skips over functions with accesses to bit fields, almost all of the program code is not optimized by InstCombine and the subsequent optimizations.

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. This is represented in figures 5.14 and 5.15.

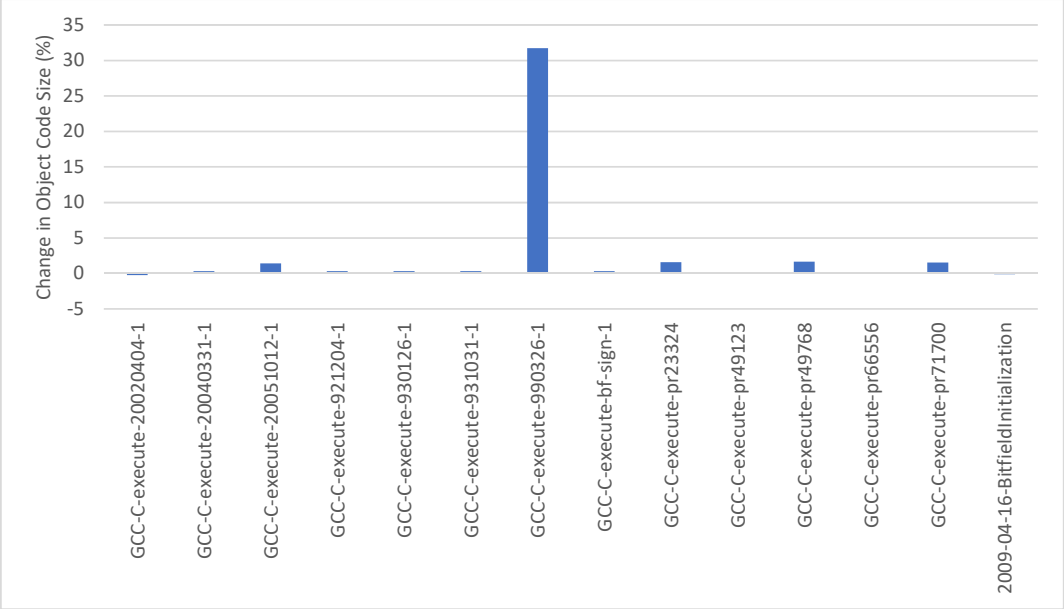

Figure 5.13: Object Code size changes in micro-benchmarks with the -O3 flag.

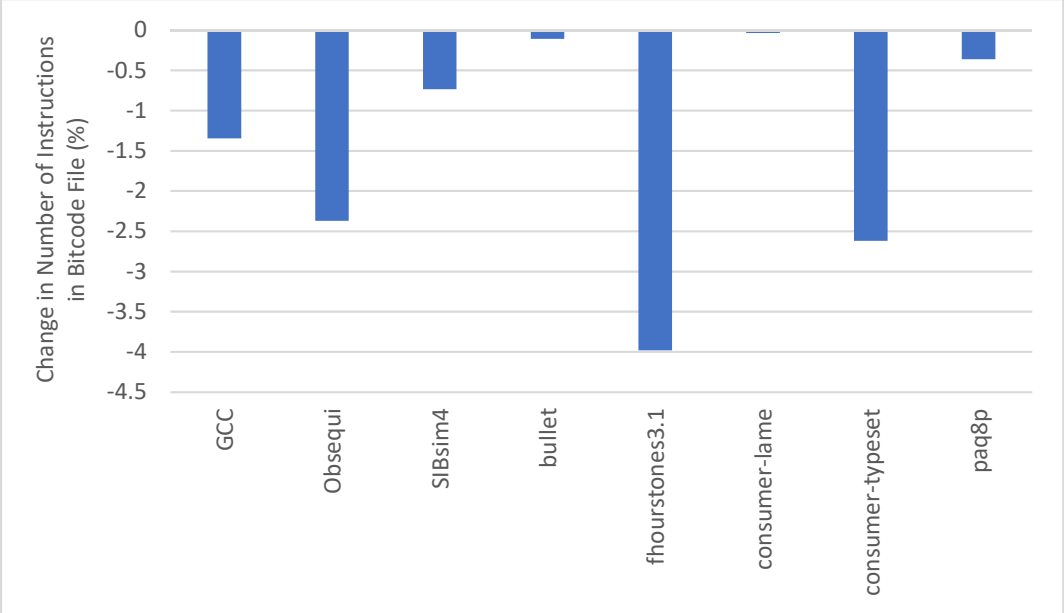

Figure 5.14: Changes in LLVM IR instructions in bitcode files in benchmarks with the -O0 flag.

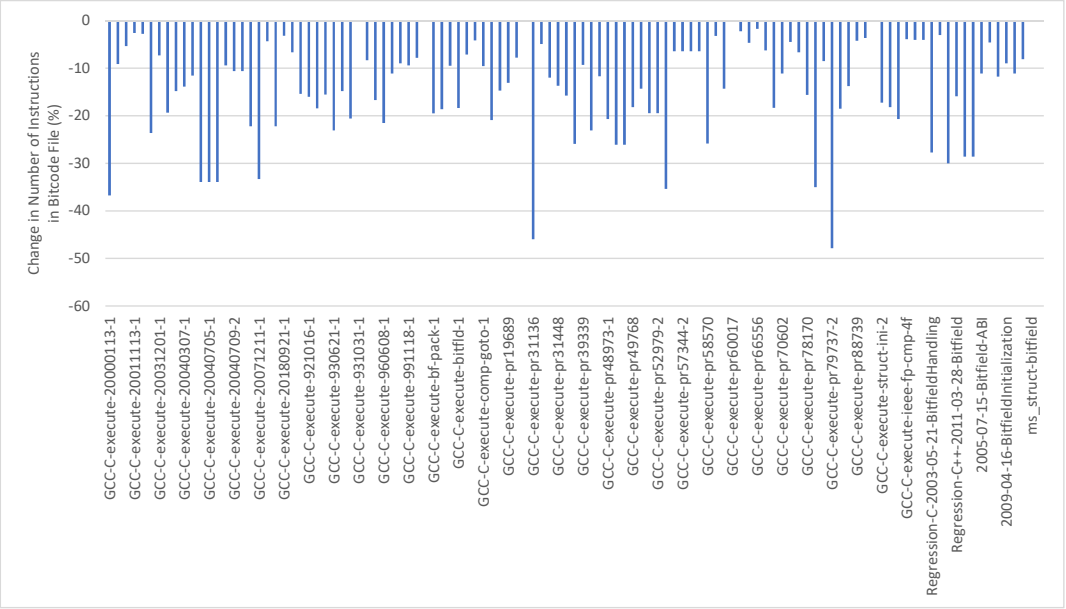

Figure 5.15: Changes in LLVM IR instructions in bitcode files in micro-benchmarks with the -O0 flag.

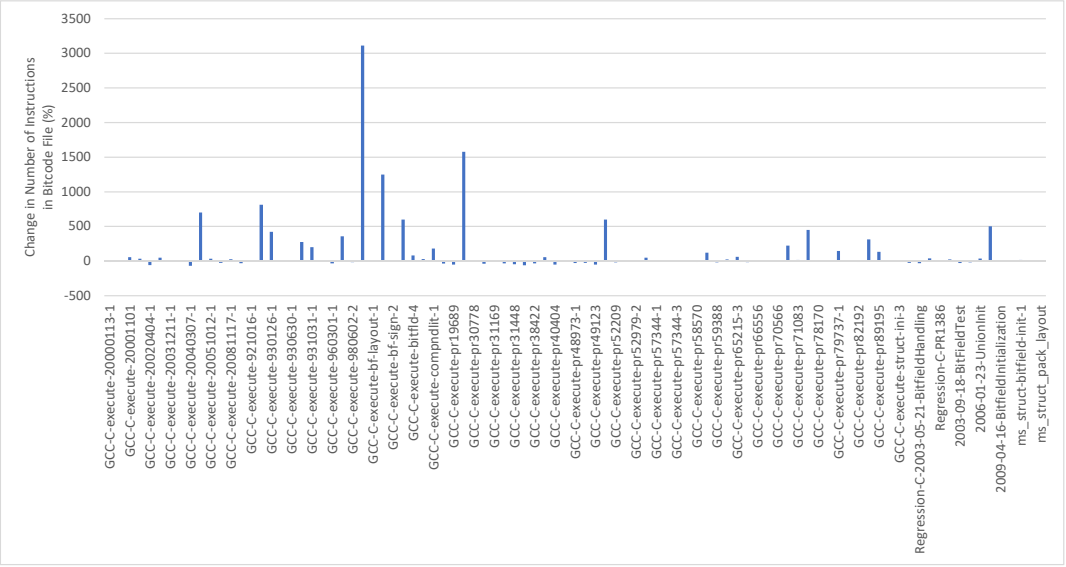

Figure 5.16: Changes in LLVM IR instructions in bitcode files in micro-benchmarks with the -O3 flag.

On the other hand, 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%, as we can see in Figure 5.16.

The reason for these discrepancies is simple: when no optimization is done, our new LLVM IR will always have less instructions simply because the operations needed to access bit fields are no longer emitted in the LLVM IR, but in the next IR the SelectionDAG. On the contrary, when we compile the tests with the -O3 flag, the InstCombine optimization fires for every function in the original compiler, while in our implementation InstCombine skips a function whenever there is a bit field access. Since InstCombine is also responsible for rewriting the IR in a canonical form on which further optimizations depend, this particular function won't be as optimized as the original. This adds up over time and we end up with a couple of micro-benchmarks with functions with a couple of instructions that should have been optimized to a single `return 0` instruction.

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% as is shown in Figure 5.17. The compilation with -O3 flag however saw an increase in one of the tests by 4.7%, as can be observed in Figure 5.18. 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.

Spills and reloads in assembly mean that there were no more available registers to store information, and that information had to be stored in memory. The reason for this rise in spills and reloads is the fact that the heuristic that is in charge of register and memory allocation is not familiar with our implementation.

The reason 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

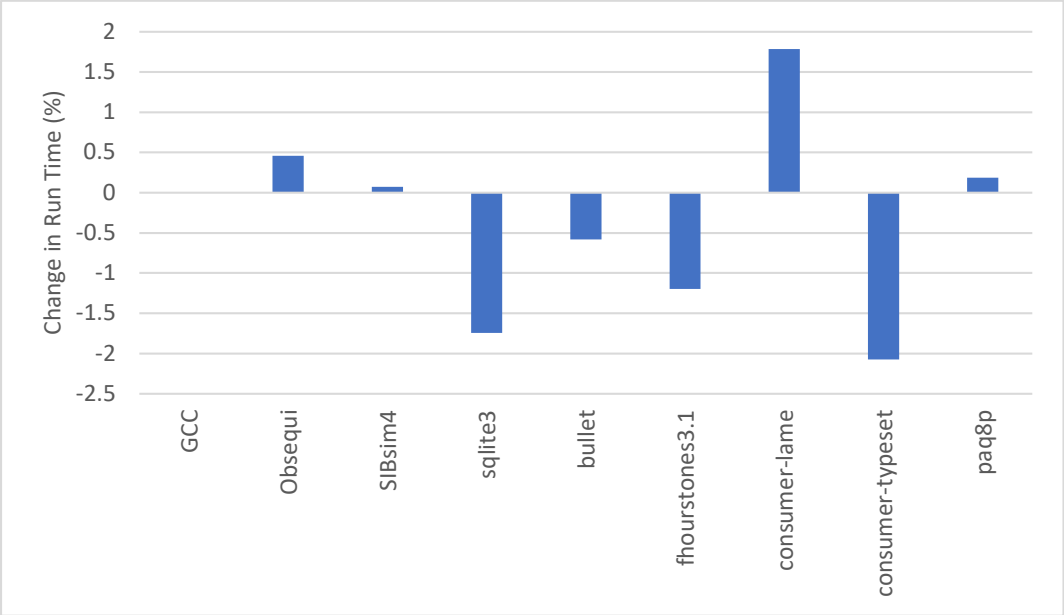

Figure 5.17: Run Time changes in benchmarks with the -O0 flag.

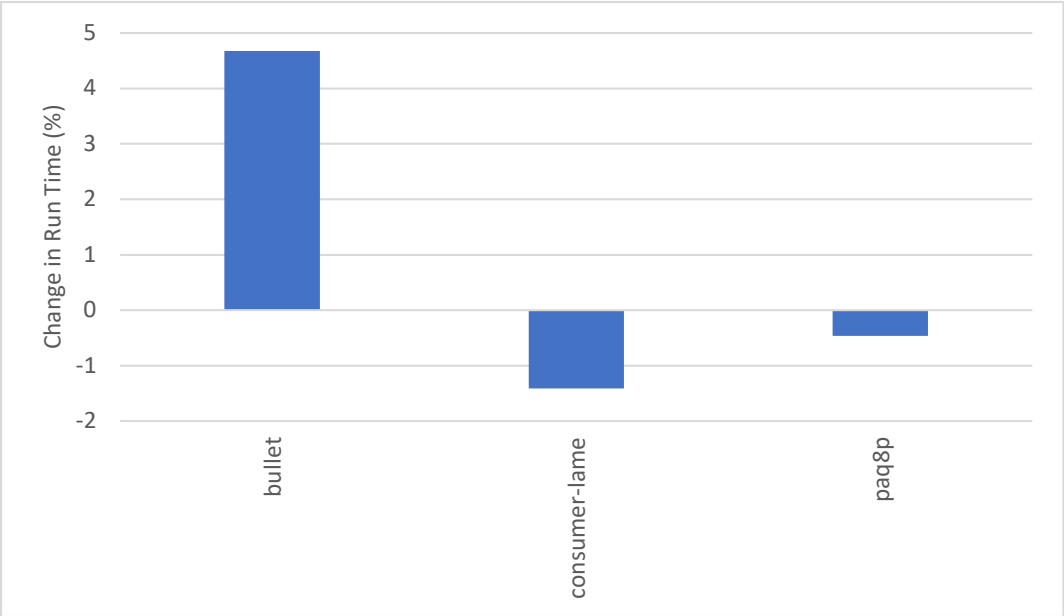

Figure 5.18: Run Time changes in benchmarks with the -O3 flag.

not a whole word, as was exemplified by the last example of section 4.2. 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.

This choice of implementation can lead to different assembly being emitted. On one hand, the number of `and`, `or` or `shift` instructions can be reduced: with a bit field of 8 bits perfectly aligned in memory we only need to load that 8-bit word, while the original implementation will load 32 bits and use `shift`, `or` and `and` instructions to get its value. A good example of this case can be observed below, which shows a copy of a bit field value with size 8 bits aligned in memory to another, also with size 8 and aligned in memory. On the left is the assembly of the original implementation and on the right the assembly of our implementation.

```
mov eax, dword ptr [rdi]
shr eax, 16
and eax, 255
mov ecx, dword ptr [rsi]
and eax, 255
shl eax, 16
and ecx, -16711681
or  ecx, eax
mov dword ptr [rsi], ecx
```

```
mov cl, byte ptr [rdi + 2]
mov byte ptr [rsi + 2], cl
```

On the other hand, sometimes the value of a bit field has to be used with another value of a bigger size, prompting an extra `movzx` or `movsx` instruction in the assembly, instructions to zero-extend or sign-extend a value, respectively. This case is seen in the code below, where a 16-bit bit field aligned in memory has to be compared with a 32-bit integer. Again, on the left is the original assembly code, with a load of 32 bits and an `and` instruction to isolate the bit field value, and on the right is the code emitted by our compiler, with just a 16-bit load but an extra zero extend instruction to be able to compare the value with the 32-bit integer.

```
mov ecx, dword ptr [rax]
and ecx, 65535
cmp ecx, 48
```

```
mov   cx, word ptr [rax]
movzx edx, cx
cmp   edx, 48
```

5.7 Summary

To conclude, this section compares all the tests of the LLVM Nightly test-suite that contain bit fields, as well as the single file program GCC compiler, when compiled with our new implementation of LLVM/-Clang with a new type for structures in the LLVM IR, and when compiled with the original LLVM/Clang implementation from which we forked.

The results are promising as they don't show a particular change in performance of the new compiler while solving the `poison` problem for bit fields and keeping the new IR more understandable. The biggest performance improvement is in the Object Code Size when it comes to number of instructions in the bitcode file. This decrease in IR instructions might have an impact in some target-independent optimizations, such as Inlining. However, only one other metric saw such an overall increase in performance: the peak memory consumption of the compiler, particularly the virtual memory size of a program. However, the increase in peak memory consumption in the "paq8p" benchmark should be investigated further.

6

Conclusions and Future Work

Contents

The representation of Undefined Behavior (UB) in a compiler has become a very important topic in compiler design. The reason being that it allows the Intermediate Representation (IR) to reflect the semantics of programming languages where UB is a common occurrence, avoid the constraining of the IR to the point where some optimizations become illegal, and also to model memory stores, dereferencing pointers, and other inherently unsafe low-level operations. Because of this, most recent compilers have a way to represent UB in their IR, and the LLVM compiler is no exception representing UB by having the `undef` keyword and the concept of `poison`.

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.

Unfortunately, the solution proposed to the bit fields problem by the new semantics was not correct. With this in mind we introduced 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. This way, it allows the programmer to easily identify the fields of the structure in the IR while also preventing the propagation of `poison` to the remaining bit fields since, before our implementation, there was a one-to-one translation between a word in the IR and a word in memory, meaning that adjacent bit fields were bundled in the same word.

Finally we compared our implementation to the LLVM/Clang implementation from which we forked by running the tests of the LLVM Nightly test-suite that contained bit fields, as well as the GCC compiler source code. We measured and compared Compile Time, Peak Memory Consumption, Object Code Size and Run Time of the tests when compiled with both implementations of the LLVM compiler, either with no optimizations (-O0 flag) or most of the "textbook" compiler optimizations we know today (-O3 flag). The results showed were on par with the performance of the original implementation of the LLVM/Clang compiler, which was what we aimed to achieve.

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. Also, and as was apparent in section 5, there are still optimizations that do not take our

implementation into account, which needs to be addressed.

It would also be important to implement the vector loading solution to this problem and see how it would fare against our implementation. Despite the IR becoming more confusing perhaps in the end the performance will improve. One thing that vector loading of bit fields has in its favor is the fact that there is no need to implement anything new to the compiler, as all it needs already exists in the IR: store the bit fields as vectors instead of integers, and change the way bit field accessing works by accessing each bit of a field individually.

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.

Bibliography

[1] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, "Taming Undefined Behavior in LLVM," *SIGPLAN Not.*, vol. 52, no. 6, pp. 633–647, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3140587.3062343

[2] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[3] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding Integer Overflow in C/C++," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 760–770. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337313

[4] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 260–275. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522728

[5] 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.

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

[7] F. E. Allen, "Control Flow Analysis," *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/390013.808479

[8] 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

[9] 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.*, vol. 13, no. 4, pp. 451–490, Oct. 1991. [Online]. Available: http://doi.acm.org/10.1145/115372.115320

[10] C. S. Ananian, “The Static Single Information Form,” 1999.

[11] 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,” *SIGPLAN Not.*, vol. 25, no. 6, pp. 257–271, Jun. 1990. [Online]. Available: http://doi.acm.org/10.1145/93548.93578

[12] P. Tu and D. Padua, “Efficient Building and Placing of Gating Functions,” *SIGPLAN Not.*, vol. 30, no. 6, pp. 47–55, Jun. 1995. [Online]. Available: http://doi.acm.org/10.1145/223428.207115

[13] 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*, vol. 15, no. 4-5, p. 526–542, 2015.

[14] C. Hathhorn, C. Ellison, and G. Roşu, “Defining the Undefinedness of C,” *SIGPLAN Not.*, vol. 50, no. 6, pp. 336–345, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2813885.2737979

[15] W. . ISO/IEC JTC 1, SC 22, “Rationale for international standard — programming languages — C. Technical Report 5.10, Intl. Org. for Standardization.” http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf, 2003, [Online; accessed 3-December-2018].

[16] M. Braun, S. Buchwald, and A. Zwinkau, “Firm - a graph-based intermediate representation,” Karlsruhe, Tech. Rep. 35, 2011.

[17] 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

[18] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, “Provably Correct Peephole Optimizations with Alive,” *SIGPLAN Not.*, vol. 50, no. 6, pp. 22–32, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2813885.2737965

[19] 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

[20] 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.

[21] S. Blazy and X. Leroy, “Mechanized Semantics for the Clight Subset of the C Language,” *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, Oct 2009. [Online]. Available: https://doi.org/10.1007/s10817-009-9148-3

[22] G. Barthe, D. Demange, and D. Pichardie, “Formal Verification of an SSA-based Middle-end for CompCert,” University works, Oct. 2011. [Online]. Available: https://hal.inria.fr/inria-00634702

[23] 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

[24] 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

[25] 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

[26] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, “Test-case reduction for c compiler bugs,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 335–346. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[27] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in c compilers,” *SIGPLAN Not.*, vol. 46, no. 6, p. 283–294, Jun. 2011. [Online]. Available: https://doi.org/10.1145/1993316.1993532