Fixing fundamental issues in LLVM IR: Introducing a byte type to solve load type punning

Student: George Mitenkov (georgemitenk0v@gmail.com)
Mentors: Nuno Lopes and Juneyoung Lee

Introduction

LLVM ecosystem provides a broad range of compiler optimisations, ranging from loops to memory. While the optimisations are covered by the FileCheck test suite, are well-defined semantically, and are peer-reviewed by several LLVM developers, miscompilations due to unsound transformations still occur. Alive2 [1], designed to verify the transformations, has recently made it easier to track bugs in LLVM, and to identify the sources of these miscompilations.

One of such sources are memory optimisations. The challenge for the compilers stems from the fact that languages like C/C++ or Rust allow programmers to have a rather low-level control of memory. This makes high-level memory optimisations hard to implement, since aliasing and memory model guarantees need to be taken into account. Recently, a new memory model for LLVM IR has been proposed in order to make memory optimisations both sound and efficient, while not blocking other integer optimisations [2]. However, there are still semantic inconsistencies in compiler-introduced load type punning.

The problem

It is common for compilers, and LLVM in particular, to transform calls to **`memcpy`** or **`memmove`** to a number of integer loads and stores of the corresponding bit width. After, load and store instructions can be optimised further. However, there are certain problems with this type punning approach.

```
call void @llvm.memcpy.p0i8.p0i8.i32(i8* %dst, i8* %src, i32 8, i1
false)
      =>
%src64 = bitcast i8* %src to i64*
%dst64 = bitcast i8* %dst to i64*
%tmp = load i64, i64* %src64, align 1
store i64 %tmp, i64* %dst64, align 1
```

Example 1 (InstCombine)
Source: https://llvm.godbolt.org/z/1G735z1ax

Firstly, semantics of **`memcpy`** [3] and **`memmove`** [4] specify that the memory is copied as-is in bytes, including the padding bits if necessary. On the other hand, in LLVM IR padding is always poison, and loading poison bits makes the whole loaded value to be poison as well [5]. If the call to **`memcpy`** or **`memmove`** is substituted with an integer load/store pair, then it is possible that the copied value turns into poison after the transformation (see example in https://alive2.llvm.org/ce/z/xoCTpH). This problem also affects other C++ functions that may be lowered to calls to **`memcpy`** or **`memmove`** and subsequently to integer load/store pairs, such as **`uninitialised_copy`** [6] (see example in https://llvm.godbolt.org/z/nGr6K4cnP). Several violations in semantic differences due to this transformation have been reported by Alive2: (1), (2), (3) and (4) as of April 1st.

The second problem comes from the fact that **`unsigned char*`** [8], used in **`memcpy`** and **`memmove`** definitions, can alias with any pointer. Hence, substituting a call to **`memcpy`** of a pointer with an integer load/store pair may fool the compiler not to see the escape of the pointer, thus breaking the alias analysis and the soundness of further optimisations. Consider the example below:

```
%src8 = bitcast i8** %src to i8*
%dst8 = bitcast i8** %dst to i8*
call void @llvm.memcpy.p0i8.p0i8.i32(i8* %dst8, i8* %src8,
                                     i32 8, i1 false)
%load = load i8*, i8** %dst
%addr = ptrtoint i8* %load to i64
ret i64 %addr
      =>
%src64 = bitcast i8** %src to i64*
%dst64 = bitcast i8** %dst to i64*
%addr = load i64, i64* %src64, align 1
store i64 %addr, i64* %dst64, align 1
ret i64 %addr
```

Example 2 (InstCombine)
Source: https://godbolt.org/z/jq9G9GMn3

In the source program, the **`ptrtoint`** instruction immediately specifies that the pointer `%load` escapes via an integer. In the target program, the **`ptrtoint`** instruction is removed, and it may happen that a value at `*%dst` may be changed without the compiler noticing it (via casting `%addr` to a pointer and storing to it).

The underlying problem is the fact that LLVM IR lowers **`unsigned char`** or similarly the recently introduced **`std::byte`** [9] to **`i8`**. While both C and C++ define these types as handles to the raw bytes of objects, LLVM IR does not have a similar type. This means that compiler-introduced type punning can break the alias analysis and miss the escaped pointer, as was reported in the bug report 37469. To briefly describe the issue, consider the following

transformations, where `a1`, `a2`, `a3` are `unsigned char` arrays of length 8 and `p`, `q`, `r` are integer pointers:

```
memcpy(a1, &p, 8);
memcpy(a2, &q, 8);
// Elementwise comparison and assignment
if ((int)a1 == (int)a2)
  a3 = a1;
else
  a3 = a2;

memcpy(&r, a3, 8);
*p = 2;
*r = 3;
```

Original program

=>

```
memcpy(a1, &p, 8);
memcpy(a2, &q, 8);

a3 = a2

memcpy(&r, a3 8);
*p = 2;
*r = 3;
```

(InstCombine 1)

=>

```
*p = 2;
*q = 3;
```

(InstCombine 2)

Example 3

The if statement in the original program is simplified to `a3 = a2` using equivalence of `y` and `(x == y) ? x : y`. After transforming the calls to `memcpy` to load/store pairs, the store to `r` is incorrectly propagated from the `memcpy` of `q`, due to implicit pointer-to-integer and integer-to-pointer casts, described in Example 2.

Solution

We propose to introduce a new **byte** type to LLVM IR to fix miscompilation issues described in the previous section and make the semantics of type punning in LLVM consistent. The byte type would have different sizes corresponding to different bit widths (e.g. `b64` for 8-byte value and `b8` for a single byte value). We further propose to make the lowering of `char`, `unsigned char`, and `std::byte` in Clang to emit the new byte type. On the SelectionDAG side, we initially plan to lower the new byte type to integer. However, a further discussion is needed to assess the need of a byte type in SelectionDAG as well.

The byte type would have the following properties:

1. The byte type would be a first-class type in LLVM IR, and treated as a union of all the other first-class types (like `std::variant`).
2. The byte type would:
    a. be allowed as a `load` or `store` value
    b. support cast operations: existing `bitcast`, and a new `cast` instructions, with the following specification
        i. `bitcast` of a byte type to the first-class type (returns poison on type punning or if any bit is poison)
        ii. `bitcast` of the first-class type to a byte type
        iii. `cast` of the byte type to the first-class type such that:

        1. if the source type is a pointer, and the target type is an integer, then the cast acts like **`ptrtoint`** and returns poison on type punning
        2. if the source type is an integer, and the target type is a pointer, then the cast acts like **`inttoptr`** and returns poison on type punning
        3. otherwise, **`bitcast`** is performed
    c. not support any kind of arithmetic operations
    d. track pointer provenance (if underlying value is a pointer)

The motivation for the generic **`cast`** instruction comes from the fact that when Clang converts the byte to an integer (like in if statement in Example 3), the underlying value of the byte may be unknown. Hence, it is impossible to deduce whether to insert **`ptrtoint`** or **`bitcast`** instructions.

A more detailed overview of the required changes and the milestones of the project are described below.

Benefits

The introduction of the new byte type has a number of benefits. We identify the following:

1. The lowering of loads of **`char`**, **`unsigned char`**, and **`std::byte`** would be fixed by emitting the new byte type instead of **`i8`**. This would open a way for fixing incorrect optimisations due to load type punning, and would be more in line with the “raw-memory access” semantics of these types.

2. The calls to **`memcpy`** and **`memmove`** would be correctly transformed into the loads and stores of the new byte type no matter what underlying type is being copied. Firstly, this solves the problem of copying the padded data, with no contamination of the loaded value due to poison bits of the padding. Also, when copying pointers implicit pointer-to-integer casts that may undermine the alias analysis would be eliminated.

3. The provenance changes due to **`inttoptr`** casts would be avoided. Before, two loads from **`i8**`** would be transformed into **`i64`** load and an **`inttoptr`** instruction, as shown in Example 4.1. The resulting provenance of `%v2` after the transformations would be full.

```
%v1 = load i8*, %p
%v2 = load i8*, %p
      =>
%v1 = load i64, %p    ; casts are dropped for brevity
%v2 = inttoptr %v1    ; too strict: %v2 has full provenance
```

Example 4.1 (InstCombine)

With the new byte type, the cast from byte to pointer would preserve the provenance since unlike integer the byte type carries one.

```
%v1 = load i8*, %p
%v2 = load i8*, %p
      =>
%v1 = load b64, %p  ; casts are dropped for brevity
%v2 = cast b64 %v1  ; provenance of %v2 is preserved
```

Example 4.2

The project

The project involves changes that involve several stakeholders, as many parts of the compiler have to be touched: frontend, optimisation passes, and the backend (SelectionDAG and GlobalISel). In this section, we present a brief description of the necessary steps of the project, as well as describe some examples of how the new byte type would fit into the LLVM IR ecosystem.

Introducing the byte type

As described in the previous section, the byte type would only support memory operations and certain casts. We therefore would need to change memory instructions to accept the new type (including **`load`**, **`store`**, **`memcpy`** and **`memmove`**).

Regarding the implementation, the byte type **`ByteType`** would inherit properties of **`llvm::Type`** class, similarly to other types within the LLVM IR.

While this overview gives an understanding of how the byte type solves the load type punning, the exact semantics of it and the operations it supports are yet to be discussed with the community. We plan to collaborate with the various stakeholders in order to choose the best possible design.

Changing the Clang frontend

Since there are numerous frontends that output LLVM IR, we choose to focus on Clang only, while making other frontend developers aware of the changes. This helps to avoid the difficulty of leveraging the changes in various frontends ourselves, and to focus on the soundness of transformations instead.

As previously discussed, we plan to change the lowering of loads of **`char`**, **`unsigned char`**, and **`std::byte`**. While the first two types are rather trivial to handle, the case of **`std::byte`** may need extra handling as it is defined as enum class [10], and may exhibit some other properties. This is currently being discussed with the Clang community.

Changing the lowering to SelectionDAG

SelectionDAG and the backend infrastructure need to support the byte type lowering as well. Currently, we propose to lower the byte type to an integer on the SelectionDAG level. We believe that a wider discussion within the community may be useful to establish alternative lowerings, such as whether the byte type is needed in SelectionDAG as well.

Fixing the unsound optimisations due to load type punning

The unsound optimisation passes would be fixed with the introduction of the new byte type. The first optimisation we would focus on is the transformation of **`memcpy`** and **`memmove`** into the loads and stores of the new byte type. This involves changing the **`instcombine`** pass. Also, we may need to change **`gvn`** pass to propagate byte's equality. We also take into account the fact that there may be other optimisations that could be blocked. Hence, we save some time to address these as well.

Introducing new optimizations to remove unnecessary instructions

Moreover, several new optimisations would need to be added to reduce the number of instructions. For example, redundant casts from bytes to integers may be folded into single integer loads, as shown in Example 5.

```
%data = load b32, %p
%v = bitcast %data to i32
      =>
%v = load i32, %p
```

Example 5

To find the optimisation opportunities, we will run the benchmarks. We will identify regressions and based on them select the candidates for optimisation. For example, we can look at the number of bitcasts from the byte type to integers, or the number of **`cast`** instructions that can be substituted with a **`bitcast`**, etc.

Evaluating soundness and performance

The equivalence of the semantics and implementation of the proposed change will be established with Alive2. Moreover, carefully-written FileCheck tests and Phabricator reviews will reduce the risks of bugs.

The change in performance that the byte type may introduce also needs to be analysed. Particularly, the change in compile-time and runtime will be investigated to account for any regression. For that, we plan to use the LLVM test suite [11].

Deliverables

Prior to the community bonding period and the announcement of the projects, I plan to deepen my understanding of the C/C++ standard with respect to memory semantics, LLVM IR memory model, the alias analysis in LLVM and the current problems with memory optimisations. I have already started looking into these issues, and plan to continue to do so. Additionally, I will continue working on some simple fixes to bugs reported in the Alive2 dashboard [12].

Community bonding period: May, 17th - June, 6th

I intend to carry on with researching and studying the semantics of the current standards and the memory models, as well as other-memory-related issues like type punning, aliasing, etc. I plan to define the semantics of the new byte type and collaborate with the community and stakeholders to make sure that the best option is chosen.

Additionally, I will carry on working on simple bugs reported by Alive2.

Phase 1: June, 7th - July, 16th

Weeks 1-2
The first two weeks are dedicated to introducing the byte type to the LLVM IR. I plan to define and implement the supported operations (e.g. making `bitcast` aware of the new type and document the changes in the LLVM LangRef)

Weeks 3-4
For the next two weeks, I plan to add support for the byte type to Clang and SelectionDAG. This involves verifying the correctness of lowerings from C/C++ to LLVM IR and from LLVM IR to SelectionDAG.

Weeks 5-6
During the last two weeks of the first phase, I will conduct performance experiments, accounting for changes in compile-time and in run-time. I plan to

analyse any sources of regression, and come up with certain optimisations (e.g. folding byte to integer cast into a single integer load) to implement in Phase 2. These two weeks also include extra time for any blocking issues.

Phase 2: July, 17th - August, 23rd

Weeks 7-8
During the first weeks of the second phase I plan to fix `instcombine` pass to change the `memcpy` and `memmove` optimisations. I will analyse the performance and validity of the changed optimisation passes with regressions tests and Alive2 respectively. I will also save some time to patch any other optimisations that may be blocked, or may need to support the byte type as well.

Weeks 9-10
During the following two weeks, I plan to continue working on the remaining optimisation fixes. I also plan to introduce new optimisations to improve any performance regression. The new optimisations will be analysed with Alive2 and benchmarked.

Week 11
Last week of the second phase is left to account for any blocking issues and unforeseen challenges. It is also dedicated for the final report writing.

Throughout the project, I plan to keep clear documentation of what has been done, actively participate in discussions within the community, and test and benchmark the new changes, so that any performance regression is taken into account straight away.

About me

I am a final year Maths and Computer Science student at Imperial College London. I am interested in performance engineering, optimisation and compilers. At university, I have successfully completed Compilers, Performance Engineering and Advanced Architecture courses, which involved developing a compiler from scratch, optimising C++ code, and understanding the impact of compiler optimisations on program execution. I have also taken a Reasoning about Programs course that taught me how important program verification can be. For my bachelor's thesis, I am developing a LLVM-based backend for NMODL compiler [repository], particularly focusing on the vectorisation aspects. Last year I participated in GSoC 2020 with the MLIR project, developing a conversion from SPIR-V dialect to LLVM IR dialect [summary]. I thus have experience with the workflow of contributing patches via Phabriacator, as well as aware of the C++ Clang coding standards. I have contributed multiple patches to the MLIR/LLVM projects, as well as have started fixing bugs reported by Alive2 [commits].

References

[1] Alive2 on GitHub: https://github.com/AliveToolkit/alive2
[2] Reconciling High-Level Optimizations and Low-Level Code in LLVM: https://web.ist.utl.pt/nuno.lopes/pubs/llvmmem-oopsla18.pdf
[3] Semantics of memcpy: http://www.cplusplus.com/reference/cstring/memcpy/
[4] Semantics of memmove: http://www.cplusplus.com/reference/cstring/memmove/
[5] Taming Undefined Behaviour in LLVM: https://www.cs.utah.edu/~regehr/papers/undef-pldi17.pdf
[6] Semantics of uninitialized_copy: https://en.cppreference.com/w/cpp/memory/uninitialized_copy
[7] LLVM LangRef: https://llvm.org/docs/LangRef.html
[8] Reference on char types: https://en.cppreference.com/w/cpp/language/types
[9] std::byte semantics: https://en.cppreference.com/w/cpp/types/byte
[10] std::byte declaration: https://github.com/llvm/llvm-project/blob/main/libcxx/include/cstddef#L87
[11] LLVM's test suite: https://llvm.org/docs/TestSuiteGuide.html
[12] Alive2 dashboard: https://web.ist.utl.pt/nuno.lopes/alive2/