Reconciling High-Level Optimizations and Low-Level Code in LLVM

JUNEYOUNG LEE, Seoul National University, Korea
CHUNG-KIL HUR, Seoul National University, Korea
RALF JUNG, MPI-SWS, Germany
ZHENGYANG LIU, University of Utah, USA
JOHN REGEHR, University of Utah, USA
NUNO P. LOPES, Microsoft Research, UK

LLVM miscompiles certain programs in C, C++, and Rust that use low-level language features such as raw pointers in Rust or conversion between integers and pointers in C or C++. The problem is that it is difficult for the compiler to implement aggressive, high-level memory optimizations while also respecting the guarantees made by the programming languages to low-level programs. A deeper problem is that the memory model for LLVM's intermediate representation (IR) is informal and the semantics of corner cases are not always clear to all compiler developers.

We developed a novel memory model for LLVM IR and formalized it. The new model requires a handful of problematic IR-level optimizations to be removed, but it also supports the addition of new optimizations that were not previously legal. We have implemented the new model and shown that it fixes known memory-model-related miscompilations without impacting the quality of generated code.

CCS Concepts: • **Software and its engineering → Semantics**; **Compilers**;

Additional Key Words and Phrases: IR Memory Model, LLVM

ACM Reference Format:
Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling High-Level Optimizations and Low-Level Code in LLVM. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 125 (November 2018), 28 pages. https://doi.org/10.1145/3276495

1 INTRODUCTION

The *memory model* for a programming language determines how programs are permitted to observe and modify storage. For example, references in Java are pointer-like in that they uniquely identify objects in memory, but programs are not allowed to construct references into the middle of objects or to fabricate references from scratch. In contrast, a truly low-level programming language, such as assembly language, allows arbitrary memory locations to be inspected and modified with no restrictions whatsoever on how addresses are computed.

C and C++ occupy an interesting niche. They are intended to be low-level languages; systems software—operating system kernels, virtual machine managers, embedded firmware, programming language runtimes, etc.—tends to be built in one or the other. To support these applications, pointers

Authors' addresses: Juneyoung Lee, Seoul National University, Korea; Chung-Kil Hur, Seoul National University, Korea; Ralf Jung, MPI-SWS, Germany; Zhengyang Liu, University of Utah, USA; John Regehr, University of Utah, USA; Nuno P. Lopes, Microsoft Research, UK.

2475-1421/2018/11-ART125
https://doi.org/10.1145/3276495

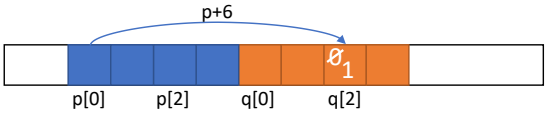


Fig. 1. In a flat memory model, storing a 1 into p[6] can overwrite the 0 in q[2]

into objects can be constructed using pointer arithmetic, and pointers may be converted to integers and integers to pointers. However, despite their low-level character, the C and C++ memory models also incorporate higher-level features. For example, the compiler is permitted to assume (with some restrictions) that a pointer to one allocated object is not used as the basis for creating a pointer to another object. The C and C++ standards committees intend for the languages to provide low-level memory access when necessary, but otherwise retain the ability to heavily optimize code as if it were high-level. This design point is not an easy one. The tension between the low-level and high-level language features is significant and causes problems for both compiler and application developers.

This paper is about sequential memory models for compiler intermediate representations (IRs). The IR level is important because it is where high-level optimizations happen. Both GCC and LLVM have memory models that are informally specified and that contain problems that lead to end-to-end miscompilation of low-level programs. Appendix A shows a C program that is miscompiled by both GCC and LLVM. However, the problems that we are attacking are broader than C and C++: Appendix B shows a low-level (but safe) Rust function that is miscompiled by LLVM. The culprit is a new bug we found in LLVM's global value numbering (GVN) optimization. GVN propagates equalities of pointers (as well as of integers) from branch conditions, replacing pointers with value-equal ones. This, however, can change the behavior of a program, since pointers that compare equal are not necessarily equivalent. The miscompilation of the C example can be traced back to a second bug we found where LLVM incorrectly assumes that `(int*)(intptr_t)p` is equal to `p`.

Fixing these miscompilations within the current IR semantics would be possible, but would necessitate disabling useful optimizations. Our main contribution, which builds on insights developed by [Kang et al. 2015], is a new, formalized IR memory model for LLVM that departs from the current design in two ways. First, it uses *deferred bounds checking* to relax restrictions on the creation of out-of-bounds pointers in such a way that useful code motion optimizations can be performed soundly. Second, it uses *twin allocation*, which formalizes the idea that the value of a pointer has to be observed directly, it cannot be guessed. Twin allocation supports aggressive optimization of LLVM-based languages in the presence of low-level code such as integer-to-pointer casts. We have adapted LLVM to the new semantics in order to show that it does not require major changes to the compiler and it also does not degrade the performance of generated code.

2 BACKGROUND: MEMORY MODELS FOR INTERMEDIATE REPRESENTATIONS

In this section we describe the design space for low-level sequential memory models and explain that existing designs are inadequate for managing the tension between low-level memory access and high-level optimizations. Then, in Section 3, we describe the basis for the new memory model for LLVM IR that we will formalize in Section 4. Our examples are written in a C-like syntax to make them easier to read, even though the scope of the paper is to specify a memory model for compiler IRs rather than for C.

2.1 Flat Memory Models

The two main questions a memory model needs to answer are (1) what is the return value of a load instruction, and (2) under what conditions is a memory-accessing instruction well-defined. A

consequence is that the memory model should define which memory locations a store instruction writes to.

For example, what does the code below print? Or, alternatively, can the assignment `p[6] = 0` change any byte of the object pointed to by `q`?

```
char *p = malloc(4);
char *q = malloc(4);
q[2] = 0;
p[6] = 1;
print(q[2]); // prints 0 or 1?
```

In a *flat* memory model, the program would print `1` if `q == p + 4`, and `0` otherwise. A flat memory model treats pointers like integers: a memory-accessing instruction can access any (unprotected) location in memory, and therefore the program is allowed to guess the location of objects (as shown in Figure 1). Some assembly languages have a flat memory model; others, such as those for machines with segmented memory, do not.

While a flat memory model is conceptually simple and is a good match for low-level programming, it hinders high-level optimizations that are routinely performed by and considered essential in modern compilers.

2.2 Data-Flow Provenance Tracking

In the previous example, we showed that the program can print either `0` or `1` depending on where the memory allocator places the allocated blocks. This dependence on the run-time behavior of the allocator overconstrains the compiler, blocking it from performing important optimizations such as store forwarding. For example, we want the compiler to be able to propagate the store `q[2] = 0` to the print instruction. Hence, the memory model needs a way to prevent the store to `p[6]` from accessing `q[2]` regardless of where `p` and `q` end up pointing at run time. For example, rules to this effect have been a part of C since C89.

Data-flow provenance tracking provides a way to prevent objects from being accessed via pointers derived from unrelated objects. The idea is that each pointer is a pair of two values: the object to which it can point to, and the memory address (or an offset within that object). It is undefined behavior (UB) to try to access memory with a pointer that is out-of-bounds of its object. This semantics is sufficient to allow the compiler to conclude that `p[6]` cannot access `q[2]`, regardless of the fact that at runtime they may end up referring to the same location.

Data-flow provenance tracking could be defined like this:

```
char *p = malloc(4);   // (val=0x10, obj=p)
char *q = malloc(4);   // (val=0x14, obj=q)
char *q2 = q + 2;      // (val=0x16, obj=q)
char *p6 = p + 6;      // (val=0x16, obj=p)

*q2 = 0;  // OK
*p6 = 1;  // UB, since out-of-bounds of obj p
print(*q2);  // can be replaced with print(0);
```

The first store through `q2` succeeds, since it is within the bounds of object `q`. The second store, however, triggers UB because the pointer is out-of-bounds of its base object (`p`), even though the program correctly guessed the address of a valid object (as shown in Figure 2). Finally, the compiler can safely propagate the store `*q2 = 0` to the print instruction since there are no well-defined store instructions in between.

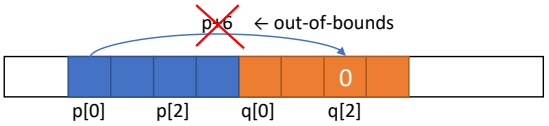


Fig. 2. In a memory model with data-flow provenance tracking, `p[6]` is not allowed to alias `q[2]`

2.3 Extending Provenance to Integers

The model we showed in the previous section does not support low-level language features like integer to pointer casts. To support this functionality, we can extend integers with provenance information: [1]

```
char *p = malloc(4);  // (val=0x10, obj=p)
char *q = (int*)0x10; // (val=0x10, obj=nil)

*q = 0; // UB, since obj=nil
if (p == q)
    *q = 1; // still UB; obj=nil

int v = (int)p;        // (val=0x10, obj=p)
int w = v + 2;         // (val=0x12, obj=p)

*(char*)w = 3;  // OK

char *r = malloc(4);   // (val=0x14, obj=r)
int x = v + (int)r;    // (val=0x24, obj=??)
int y = x - (int)r;    // (val=0x10, obj=??)
```

In this model, each integer and pointer variable tracks a numeric value plus the object it refers to, or `nil` if none. As in the previous model, addresses of objects, even if stored as integer variables, need to be derived from an object. Hence, the stores through `q` are UB. The accesses through `w` are well-defined since the value of this integer variable derives (data-flow wise) from a valid object. The last lines of the example show that provenance tracking breaks down when doing integer arithmetic operations. It is hard to assign meaningful semantics to cases like these.

A drawback of this model—fatal in practice—is that it blocks many integer optimizations, such as propagation of equalities as done by, e.g., global value numbering (GVN) or range analysis. For example, transforming "`(a == b) ? a : b`" into "`b`" is incorrect in this model: even if two integer variables compare equal, they may still have different provenances. We give a more complete example to demonstrate the problem:

```
char *p = malloc(4);  // (val=0x10, obj=p)
char *q = malloc(4);  // (val=0x14, obj=q)
int v = (int)p + 4;   // (val=0x14, obj=p)
int w = (int)q;       // (val=0x14, obj=q)

if (v == w)
    *(int*)w = 2;
```

In this program, `v` and `w` happen to have the same value, but they differ in their provenance. Hence it is not safe to replace "`*(int*)w = 2`" with "`*(int*)v = 2`". Doing so would introduce UB since "`v`" is only allowed to access object `p` and its offset is out-of-bounds. However, this sort

[1] We will use `int` to represent an integer type that is sufficiently large to hold a pointer for brevity.

of equality propagation is routinely done by GVN. In fact, a transformation similar to this one performed by GVN was responsible for miscompiling the Rust code shown in Appendix B.

2.4 Wildcard Provenance

The previous memory model has the advantage of supporting low-level operations and enabling high-level memory optimizations. However, since integer variables now carry provenance, it makes some integer optimizations unsound. One way to solve this problem is to remove provenance from integer variables, as follows:

```
char *p = malloc(4);    // (val=0x10, obj=p)
char *q = malloc(4);    // (val=0x14, obj=q)
int v = (int)p + 4;     // (val=0x14)
int w = (int)q;         // (val=0x14)

if (v == w) {
    char *r = (int*)w;  // (val=0x14, obj=*)
    *r = 2;
}
```

The differences to the example in the previous section are that (1) integers only carry a numeric value, and (2) a pointer obtained by casting from an integer can access any object (represented with a `*`). Therefore, `v` and `w` can be used interchangeably, and GVN for integers becomes sound again. This model has a major disadvantage: precise alias analysis becomes very difficult as soon as a single integer-to-pointer cast has been performed by the program being compiled. In the next sections we explore how to recover precision.

2.5 Inbounds Pointers

In the previous section, we presented a model with wildcard provenance; this works, but it impedes precise alias analysis. In this section we explain the model currently used by LLVM where pointer arithmetic is optionally "inbounds," allowing some precision to be recovered by making out-of-bounds pointer arithmetic undefined:

```
char *p = malloc(4);  // (val=0x10, obj=p)
char *q = foo(p);     // (val=0x13, obj=p)
char *r = q +inb 2;   // poison: 0x15 is out of bounds of p

p[1] = 0;
*r = 1;               // UB
print(p[1]);          // prints 0 or 1?
```

When doing inbounds pointer arithmetic (`+inb`), the base pointer and the result must be within bounds of the same object (or one past its end). This is not the case in `r`, hence the result of the operation is **poison**, which then makes the dereference of this pointer UB [LangRef 2018].

Even if the compiler does not know the value of `q`, because of the inbounds pointer arithmetic it now knows that the minimum offset of `r` has to be two, since both `q` and `r` have to be in bounds of the same object (i.e., $0 \leq o_q \leq n$ and $0 \leq o_r \leq n$, with o_q and o_q being the offsets of `q` and `r` respectively within the object, and n the object size). Since the access to `p[1]` only accesses offset one of an object, and `*r` can only access offset two or beyond, the compiler can conclude these accesses do not alias (and that the program prints `0` always).

3 A MEMORY MODEL FOR LLVM

This section informally describes a modified IR-level memory model that: enables high-level optimizations while still supporting low-level code; does not restrict movement of pointer arithmetic instructions (which remain pure functions); and, does not inhibit any standard integer optimizations. Section 4 formalizes the new model.

3.1 Deferred Bounds Checking

A drawback of LLVM's current inbounds pointer checking is that it prevents reordering of pointer arithmetic instructions and allocation functions:

```
char *p = malloc(4);   // (val=0x10, obj=p)
char *q = malloc(4);   // (val=0x14, obj=q)

char *r = (char*)((int)p + 5);  // (val=0x15, obj=*)
char *s = r +inb 1;             // (val=0x16, obj=q)
*s = 0; // OK
```

In this example, `s` is a valid pointer (i.e., it is in bounds of an object). However, if we move the definitions of `r` and `s` across that of `q`, `s` becomes out-of-bounds, and thus gets assigned **poison**.[2]

Constraining the movement of instructions is not desirable, since it inhibits optimizations like code hoisting. LLVM, as it turns out, freely moves pointer arithmetic instructions around. This is unsound. Our new model fixes the problem by instead using *deferred* bounds checking, in contrast with LLVM's current *immediate* bounds checking.

In deferred bounds checking, we allow out-of-bounds pointers to be created and manipulated; undefined behavior is only triggered when such a pointer is dereferenced. It is now OK to reorder pointer arithmetic across allocation functions in the previous example:

```
char *p = malloc(4);            // (val=0x10, obj=p)

char *r = (char*)((int)p + 5);  // (val=0x15, obj=*)
char *s = r +inb 1;             // (val=0x16, obj=*, inb={0x15,0x16})

char *q = malloc(4);            // (val=0x14, obj=q)

*s = 0; // OK since 0x15 and 0x16 are inbounds of same object
```

For pointers with `obj=*`, we now track a set of addresses that have to be within bounds of the same object when the pointer is dereferenced. On every inbounds pointer arithmetic operation, we record in the `inb` field the base pointer as well as the resulting pointer. A memory access operation is UB if not all the addresses in `inb` are within bounds of the same object. Therefore, the inbounds check is *delayed* until the pointer is dereferenced. While deferred bounds checking achieves the same effect as immediate bounds checks, it allows free movement of pointer arithmetic instructions since they now do not depend on the memory state.

Precision could be further increased by replacing `*` provenance with the object(s) the cast pointer refers to. In the first example above we could define `r` to have value `(val=0x15, obj=q)` instead. However, this is also a form of immediate bounds checking with the same movement restriction. Moreover, some addresses are in bounds of two objects, like `0x14` in the example (corresponds to `p + 4` and `q`), adding further complexity to the model. Therefore, we do not use these semantics.

[2]Note that while this example is not correct in memory models with data-flow provenance tracking, it is ok in our model. Even though we build a pointer into `q` based on `p`, this might be the result of the compiler propagating an equality that established that `p == q + 4`.

3.2 Preventing Address Guessing

This section introduces *twin allocation*, a technique that allows our model to prevent a program from guessing addresses of objects without data-flow provenance tracking.

A problem with wildcard provenance is that a pointer formed out of an integer can access any object. Consequently, a program may be able to guess the address of any object and access it. This makes precise alias analysis very difficult.

A simple idea to prevent guessing is to exploit the fact that allocation functions return a non-deterministic value:

```
char *p = malloc(4);  // (val=*, obj=p)
char *q = 0x10;
*q = 0; // UB if val(p) != 0x10
```

This program can guess the address of `p` in a possible execution where `malloc` returns `0x10`. However, there is at least one execution where the program fails to guess the address of `p` (e.g., `malloc` returns `0x20`), and so the program would trigger UB. Since there is at least one execution where the program would trigger UB, the compiler can assume `q` cannot alias with `p` (or with any object at all).

Even with non-deterministic allocations, a program can still (desirably) observe the address of an object such that a pointer created by casting from an integer can alias with that object, e.g.:

```
char *p = malloc(4);  // (val=*, obj=p)
*p = 0;
int v = 0x10;
if ((int)p == v)
    *(int*)v = 1;
print(*p); // can print 0 or 1
```

This program is well-defined and may print `0` or `1`, depending on the return value of `malloc`. The comparison is sufficient for the program to observe the address of object `p`, and so `*(int*)v = 1` will not trigger UB in any execution.

However, this semantics still has a caveat: it allows programs to guess addresses through a "side-channel leak." The leak happens when the memory only has a single address left where a new object can be allocated. For example, assume that a system has an 8-bit heap segment as well as 8-bit pointers, that heap address `0x00` is legal, and that the allocations below succeed:

```
char *p = malloc(0x80);
char *q = malloc(0x80);

*q = 0;
int v = ((int)p == 0x00) ? 0x80 : 0x00;
*(char*)v = 1;

print(*q); // prints 1
```

Since each heap cell is half the size of the address space, there are only two possible heap configurations: p-first or q-first. Therefore, a single test allows the program to guess the address of `q` without having to explicitly observe it. In other words, when memory is finite, returning a non-deterministic value from allocation functions is not sufficient to prevent programs from guessing addresses.

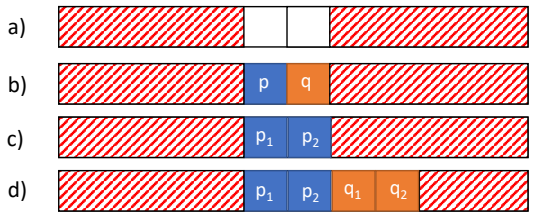


Fig. 3. Memory configuration: (a) almost full with only two bytes left, (b) after allocating p and q in (a), (c) after allocating p with twin allocation semantics in (a), (d) alternative configuration where twin allocation had enough space for both objects.

Our solution is to change allocation functions to reserve (at least) *two* blocks instead of a single one, as it happens in run-time implementations. We call this technique *twin allocation*, and we use it to formalize the notion that a program cannot guess the address of an object.

We will informally explain the concept of twin allocation with the following example:

```
char *p = malloc(1);
char *q = malloc(1);

*q = 0;
int v = (int)p + 1;  // equal to q?
*(char*)v = 1;

print(*q); // prints 0 or 1?
```

Since the address of q was not observed, we would like the compiler to be able to conclude that the program can only print 0. However, as we have seen previously, if the memory is full, observing the address of one object may implicitly disclose the address of another object. In Figure 3(a) we show a possible memory configuration before our allocations, and (b) shows the configuration after allocating p and q.

With twin allocation, each allocation function reserves at least two blocks. Non-deterministically, one of the blocks is used and its address is returned, and the remaining blocks are marked as unreachable (i.e., it is UB to access those memory regions). By reserving two blocks, we guarantee there is enough non-determinism left such that the program cannot guess the address of a block even if the memory is full.

In Figure 3(c), we show that with twin allocation our previous example would simply run out of memory, and thus the program cannot continue and try to guess the address of q. In (d) we show another memory configuration where the space left was just enough to allocate two blocks for each allocation. Since we have two blocks per object, `malloc` can still return one of the two addresses non-deterministically, effectively inhibiting the program from guessing the address of an object. Even if a program is able to guess the address of, say, p_1 (and even p_2 as well), it is not able to guess which of the remaining addresses points to q: It cannot know which of q_1 and q_2 was used.

3.3 Summary

We have informally presented our memory model for LLVM IR. To support both high-level optimizations and low-level code, we split pointers into two categories. First, logical pointers, derived from allocation sites, for which we do data-flow dependence tracking. That is, a pointer q obtained by pointer arithmetic operations from p (e.g., `q = p + x`) can only access the same object as p. Second, physical pointers, derived from integer-to-pointer casts, for which we do not do data-flow

$$
\begin{array}{ll}
\mathrm{Num}(sz) & ::= \{\, i \mid 0 \le i < 2^{sz} \,\} \\
\mathrm{Time} & ::= \mathbb{N} \\
\mathrm{BlockID} & ::= \mathbb{N} \\
\mathrm{CallID} & ::= \mathbb{N} \\
\mathrm{Mem} & ::= \mathrm{Time} \times (\mathrm{BlockID} \rightharpoonup \mathrm{Block}) \times (\mathrm{CallID} \rightharpoonup \mathrm{Time} \uplus \{\mathbf{None}\}) \\
\mathrm{AddrSpace} & ::= \mathbb{N} \\
\mathrm{Block} & ::= \{\, (t, r, n, a, c, P) \mid t \in \{\, \texttt{stack}, \texttt{heap}, \texttt{global}, \texttt{function} \,\} \wedge r \in (\mathrm{Time} \times (\mathrm{Time} \uplus \{\infty\})) \wedge \\
 & \qquad n \in \mathbb{N} \wedge a \in \mathbb{N} \wedge c \in \mathrm{Byte}^n \wedge P \in (\mathrm{AddrSpace} \rightharpoonup \mathrm{Num}(64)^{N+1}) \,\} \\
\mathrm{LogAddr}(s) & ::= \{\, \mathrm{Log}(l, o, s) \mid l \in \mathrm{BlockID} \wedge o \in \mathrm{Num}(\mathrm{ptrsz}(s)) \,\} \\
\mathrm{PhyAddr}(s) & ::= \{\, \mathrm{Phy}(o, s, I, cid) \mid o \in \mathrm{Num}(\mathrm{ptrsz}(s)) \wedge I \subset \mathrm{Num}(\mathrm{ptrsz}(s)) \wedge cid \in \mathrm{CallID} \uplus \{\mathbf{None}\} \,\} \\
\mathrm{Addr}(s) & ::= \mathrm{LogAddr}(s) \uplus \mathrm{PhyAddr}(s) \\
[\![\mathbf{i}sz]\!] & ::= \mathrm{Num}(sz) \uplus \{\, \mathbf{poison} \,\} \\
[\![\langle sz \times ty \rangle]\!] & ::= \{0, \ldots, sz-1\} \to [\![ty]\!] \\
[\![ty*]\!] & ::= \mathrm{Addr}(0) \uplus \{\, \mathbf{poison} \,\} \\
\mathrm{Name} & ::= \{\, \texttt{\%x}, \texttt{\%y}, \ldots \,\} \\
\mathrm{Reg} & ::= \mathrm{Name} \to \{\, (ty, v) \mid v \in [\![ty]\!] \,\} \\
\mathrm{Byte} & ::= \mathrm{Bit}^8 \\
\mathrm{Bit} & ::= [\![\mathbf{i}1]\!] \uplus \mathrm{AddrBit} \\
\mathrm{AddrBit} & ::= \{\, (p, i) \mid \exists s.\ p \in \mathrm{Addr}(s) \wedge (0 \le i < \mathrm{ptrsz}(s)) \,\}
\end{array}
$$

Fig. 4. Definitions. $\mathrm{ptrsz}(s)$ is the pointer size (in bits) for a given address space s (e.g., 64). The set of all possible values of a type ty is given by $[\![ty]\!]$.

dependence tracking, since that would block standard integer optimizations such as equality propagation. We employ two new techniques to recover precision instead: delayed bounds checking (to restrict the set of objects a pointer can point to, while keeping pointer arithmetic operations pure), and twin memory allocation (to prevent address guessing).

4 SEMANTICS AND TRANSFORMATIONS

In this section, we present a formal view of the modified memory model for LLVM that we informally presented in Section 3. Our top-level design goal was to support low-level operations required by C and C++, such as casting between integers and pointers, while also enabling high-level memory optimizations. Additional goals were: not interfering with integer optimizations, not constraining opportunities for code motion, not requiring major changes to make LLVM conform to the new model, and finally, avoiding significant regressions in compile time and in quality of generated code.

4.1 Logical and Physical Pointers

As we saw in Section 2.4, we have two types of pointers. *Logical* pointers are obtained by calling an allocation function or by doing pointer arithmetic on a logical pointer. In Section 2.4 these are the pointers with provenance of one object, e.g., `(val=0x10, obj=p)`. The second type of pointer is the *physical* pointer, which is the result of an integer-to-pointer cast. In Section 2.4, these are the pointers with wildcard provenance, e.g., `(val=0x10, obj=*)`; they can access any object.

In Fig. 4 we show the definitions for our model. Logical and physical pointers are represented, respectively, by $\mathrm{Log}(l, o, s)$ and $\mathrm{Phy}(o, s, I, cid)$.

Logical Pointers. A logical pointer $\mathrm{Log}(l, o, s)$ consists of a logical block id l, an offset o within the block, and the address space s it corresponds to (explained later in Section 4.2). A logical pointer corresponds to the address $P + o$ on the physical machine, where P is the base address of block l.

Logical pointers realize the rule that pointers derived from one object can never be used to modify other objects. This is achieved by making it impossible to change the value of l: Pointer arithmetic only affects the offset o.

Physical Pointers. As we have seen in Section 2.4, tracking objects in a pointer obtained by an integer-to-pointer cast is not viable because (1) some addresses may be within bounds of multiple objects, and (2) it prevents reordering of instructions. Hence we introduce physical pointers, roughly corresponding to pointers in a flat memory model.

A physical pointer $\mathsf{Phy}(o, s, I, cid)$ consists of an offset o within the address space s (i.e., the physical address), as well as two additional fields I and cid to restrict the set of objects the pointer can access. This allows us to recover alias analysis precision, and it enables several of the optimizations allowed by the C and C++ standards. Field I is a set of physical addresses that corresponds to the `inb` field we used to specify deferred bounds checking in Section 3.1. When the pointer is dereferenced, each address in I must be inbounds of the same object as o. Field cid is a call id, which corresponds to the time stamp when the pointer was passed as argument to a function, or **None** if the pointer did not originate from an argument. The intent is to show that pointers received as arguments do not alias locally allocated objects, e.g.:

```
int f(int *p) {
    int a = 0;
    if (&a == p)
        *p = 1;
    return a;  // returns 0 or 1?
}
```

Since a physical pointer can access any object, if there was no call id restriction, this function could return either `0` or `1`. However, since `p` has a call id of a function call still on the call stack, it cannot access any object created after the call.

The motivation to have an indirection in the time stamp of the call time in the pointer (cid indexes in memory M to retrieve the time stamp) is to support escaping pointers. Escaped physical pointers should behave as their cids being **None** after termination of the function call. This supports moving function calls across other function calls. If a function stores a pointer received by argument in a global variable, we did not want to have to record that fact and change all such pointers when the function returns. This way we only need to change the mapping between cid and time stamp on function return (set it to **None**).

4.2 Address Spaces

LLVM uses address spaces to represent distinct memories, and our memory model also supports this feature. For example, a machine might use one address space for the CPU and another for the GPU, or one address space for code and another for data. Since both memories may have overlapping address ranges (e.g., they may both use addresses in the range $[0, 2^{64})$), the address space field in pointers is used to disambiguate between the two.

The main memory of the CPU is assigned the address space zero. It is possible that a physical memory region is mapped into multiple address spaces. In this case the application can use an address cast instruction to convert pointers between address spaces.

A consequence of possible overlapping of address spaces is that pointers belonging to different address spaces may alias. To improve precision of alias analysis, we parameterize our model by the overlap.

$$\frac{n = [\![len]\!]_R \quad c = \mathbf{i}(8 \times n)\Downarrow(\mathbf{poison}) \quad P \text{ unallocated physical addresses}, \quad l \text{ fresh} \quad m' = m[l \mapsto (\mathsf{heap}, (\tau_{cur}, \infty), n, a, c, P)]}{R, (\tau_{cur}, m, C) \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{Log}(l, 0, 0)], (\tau_{cur} + 1, m', C)} \quad (\iota = \text{“}r = \textbf{call i8}* \textbf{ malloc}(\textbf{i64 } len)\text{”})\ \textsc{malloc}$$

$$\frac{\mathsf{Log}(l, 0, 0) = [\![ptr]\!]_R \quad m(l) = (\mathsf{heap}, (b, \infty), n, a, c, P) \quad m' = m[l \mapsto (\mathsf{heap}, (b, \tau_{cur}), n, a, c, P)]}{R, (\tau_{cur}, m, C) \overset{\iota}{\hookrightarrow} R, (\tau_{cur} + 1, m', C)} \quad (\iota = \text{“}\textbf{call void free}(\textbf{i8}* ptr)\text{”})\ \textsc{free-logical}$$

$$\frac{\mathsf{Log}(l, o, s) = [\![op]\!]_R \quad \mathsf{cast2int}_M(l, o, s) = j}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto j\%2^{sz}], M} \quad (\iota = \text{“}r = \textbf{ptrtoint } ty* \ op \textbf{ to i}sz\text{”})\ \textsc{ptrtoint-logical}$$

$$\frac{i = [\![op]\!]_R \quad p = \mathsf{Phy}(i, 0, \emptyset, \mathbf{None})}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto p], M} \quad (\iota = \text{“}r = \textbf{inttoptr i}64 \ op \textbf{ to } ty*\text{”})\ \textsc{inttoptr}$$

$$\frac{\mathsf{Log}(l, o_1, s) = [\![op_1]\!]_R \quad \mathsf{Log}(l, o_2, s) = [\![op_2]\!]_R}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 = o_2)], M} \quad (\iota = \text{“}r = \textbf{icmp eq } ty* \ op_1 \ op_2\text{”})\ \textsc{icmp-ptr-logical}$$

$$\frac{\mathsf{Log}(l_1, o_1, s) = [\![op_1]\!]_R \quad \mathsf{Log}(l_2, o_2, s) = [\![op_2]\!]_R \quad l_1 \neq l_2}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \mathbf{false}], M} \quad (\iota = \text{“}r = \textbf{icmp eq } ty* \ op_1 \ op_2\text{”})\ \textsc{icmp-ptr-logical'}$$

$$\frac{\mathsf{Phy}(o_1, s, I_1, cid_1) = [\![op_1]\!]_R \quad \mathsf{Phy}(o_2, s, I_2, cid_2) = [\![op_2]\!]_R}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 = o_2)], M} \quad (\iota = \text{“}r = \textbf{icmp eq } ty* \ op_1 \ op_2\text{”})\ \textsc{icmp-ptr-physical}$$

$$\frac{\mathsf{Phy}(o_1, s, I_1, cid_1) = [\![op_1]\!]_R \quad \mathsf{Phy}(o_2, s, I_2, cid_2) = [\![op_2]\!]_R}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 \leq_u o_2)], M} \quad (\iota = \text{“}r = \textbf{icmp ule } ty* \ op_1 \ op_2\text{”})\ \textsc{icmp-ule-ptr-physical}$$

$$\frac{\mathsf{Log}(l_1, o_1, s) = [\![op_1]\!]_R \quad \mathsf{Log}(l_2, o_2, s) = [\![op_2]\!]_R \quad m(l_1) = (t_1, r_1, n_1, a_1, c_1, P_1) \quad m(l_2) = (t_2, r_2, n_2, a_2, c_2, P_2) \quad l_1 \neq l_2 \quad \big((o_1 = n_1 \wedge o_2 = 0) \ \vee o_1 > n_1 \vee (o_1 = 0 \wedge o_2 = n_2) \ \vee \ o_2 > n_2 \vee r_1, r_2 \text{ disjoint}\big)}{R, (\tau_{cur}, m, C) \overset{\iota}{\hookrightarrow} R[r \mapsto \mathbf{true}], (\tau_{cur}, m, C)} \quad (\iota = \text{“}r = \textbf{icmp eq } ty* \ op_1 \ op_2\text{”})\ \textsc{icmp-ptr-logical-nondet-true}$$

$$\frac{\mathsf{Log}(l, o_1, s) = [\![op_1]\!]_R \quad \mathsf{Log}(l, o_2, s) = [\![op_2]\!]_R}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 \leq_u o_2)], M} \quad (\iota = \text{“}r = \textbf{icmp ule } ty* \ op_1 \ op_2\text{”})\ \textsc{icmp-ptr-ule-logical}$$

$$\frac{\mathsf{Log}(l, o, s) = [\![op_1]\!]_R \quad i = [\![op_2]\!]_R \quad o' = (o + \mathsf{bytewidth}(ty) * i) \,\%\, 2^{\mathsf{ptrsz}(s)}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{Log}(l, o', s)], M} \quad (\iota = \text{“}r = \textbf{gep } ty* \ op_1 \ \textbf{i}sz \ op_2\text{”})\ \textsc{gep-logical}$$

$$\frac{\mathsf{Phy}(o, s, I, cid) = [\![op_1]\!]_R \quad i = [\![op_2]\!]_R \quad o' = (o + \mathsf{bytewidth}(ty) * i) \,\%\, 2^{\mathsf{ptrsz}(s)}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{Phy}(o', s, I, cid)], M} \quad (\iota = \text{“}r = \textbf{gep } ty* \ op_1 \ \textbf{i}sz \ op_2\text{”})\ \textsc{gep-physical}$$

$$\frac{\mathsf{Log}(l, o, s) = [\![op_1]\!]_R \quad o' = o + \mathsf{bytewidth}(ty) * i \quad i = [\![op_2]\!]_R \quad \mathsf{inbounds}_M(l, o) \quad \mathsf{inbounds}_M(l, o')}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{Log}(l, o', s)], M} \quad (\iota = \text{“}r = \textbf{gep inbounds } ty* \ op_1 \ \textbf{i}sz \ op_2\text{”})\ \textsc{gep-inbounds-logical}$$

$$\frac{\mathsf{Phy}(o, s, I, cid) = [\![op_1]\!]_R \quad i = [\![op_2]\!]_R \quad o' = o + \mathsf{bytewidth}(ty) * i \quad 0 \leq o' < 2^{\mathsf{ptrsz}(s)}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{Phy}(o', s, I \cup \{\, o, o' \,\}, cid)], M} \quad (\iota = \text{“}r = \textbf{gep inbounds } ty* \ op_1 \ \textbf{i}sz \ op_2\text{”})\ \textsc{gep-inbounds-physical}$$

$$\frac{\mathsf{Log}(l, o_1, s) = [\![op_1]\!]_R \quad \mathsf{Log}(l, o_2, s) = [\![op_2]\!]_R \quad i = (o_1 - o_2) \,\%\, 2^{\mathsf{ptrsz}(s)}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto i], M} \quad (\iota = \text{“}r = \textbf{psub } ty* \ op1, \ op2\text{”})\ \textsc{psub-logical}$$

$$\frac{\mathsf{Log}(l_1, o_1, s) = [\![op_1]\!]_R \quad \mathsf{Log}(l_2, o_2, s) = [\![op_2]\!]_R \quad l_1 \neq l_2}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \mathbf{poison}], M} \quad (\iota = \text{“}r = \textbf{psub } ty* \ op1, \ op2\text{”})\ \textsc{psub-logical-poison}$$

$$\frac{\mathsf{Phy}(o_1, s, I_1, cid_1) = [\![op_1]\!]_R \quad \mathsf{Phy}(o_2, s, I_2, cid_2) = [\![op_2]\!]_R \quad i = (o_1 - o_2) \,\%\, 2^{\mathsf{ptrsz}(s)}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto i], M} \quad (\iota = \text{“}r = \textbf{psub } ty* \ op1, \ op2\text{”})\ \textsc{psub-physical}$$

Fig. 5. Selected rules of our operational semantics

4.3 Memory Blocks

We define memory $M = \text{Time} \times (\text{BlockID} \nrightarrow \text{Block}) \times (\text{CallID} \nrightarrow \text{Time} \uplus \{\textbf{None}\})$ as a triple of a time stamp, a map from logical block ids to memory blocks, and a map recording the time stamp of each function call (indexed on *cid* of physical pointers). When a new memory block is created (e.g., with **malloc** or **alloca**) or deallocated, the time stamp is incremented by one.

A memory block is a tuple (t, r, n, a, c, P), where t is the block type (e.g., stack or heap allocated), r is the life range of the block, n is the block size in bytes, a is the alignment, c the contents of the block (the actual data), and P has the addresses of the block.

When a block is deallocated (e.g., with **free** or on function exit), it is not deleted from memory. [3] Instead, we set the end of the lifetime range to the current memory time stamp, and increment it as well. FREE-LOGICAL in Fig. 5 shows the semantics of **free**. If a physical pointer $\mathsf{Phy}(o, s, I, cid)$ is given to **free**, it is equivalent to freeing a dereferenceable block whose base address is o. Double-freeing a block or **free** with a logical pointer with non-zero offset is UB. **free** with NULL is no-op.

Memory allocation functions reserve at least two blocks: The block actually observed by the program, and N additional *twin blocks*. The number N of twin blocks is a parameter of the semantics. (We will discuss in Section 4.12 why $N = 1$ might not be enough.) The block's base addresses in address space s are stored in $P(s)$, a sequence of physical addresses: $P(s)_0$ is the actual base address used and observed by the program, while the remaining addresses in the sequence are the base addresses of the twin blocks.

Crucially, we maintain the invariant that for P, P' of any pair of different live (non-deallocated) blocks, the address ranges $[P(s)_i, P(s)_i + n)$ and $[P'(s)_j, P'(s)_j + n)$ are disjoint. Thus, **malloc** reserves space for both the block and its twins (Fig. 5). The rest of the semantics only depends on $P(s)_0$ and ignores the remaining base addresses.

In Fig. 5, notation $[\![op]\!]_R$ represents the evaluation of an instruction's operand:

$$[\![v]\!]_R = \begin{cases} R(v) & \text{if } v \text{ is a register} \\ v & \text{if } v \text{ is a constant or } v = \textbf{poison} \end{cases}$$

4.4 Pointer Arithmetic

LLVM IR has a single instruction for pointer arithmetic, **getelementptr**, or **gep** for short. In Fig. 5 we show the semantics of several cases. For a logical pointer, the result is also a logical pointer where only the offset is updated (GEP-LOGICAL). Likewise for physical pointers (GEP-PHYSICAL).

Function $\text{inbounds}_M(l, o)$ checks if a given offset o is within bounds of object l in memory $M = (\tau, m, C)$. If $m(l) = (t, r, n, a, c, P)$, $\text{inbounds}_M(l, o)$ is true iff $0 \le o \le n$.

When **gep** has the **inbounds** tag (e.g., when compiling C/C++ code or most cases in safe languages), the compiler can assume that the input pointer is valid (within bounds of some object or else one element past the end) and also that the resulting pointer is valid. For logical pointers, the bounds checking is immediate (GEP-INBOUNDS-LOGICAL). If either of the inbounds conditions fails, the result is **poison** (not shown).

If the pointer is physical, we use deferred bounds checking: input and output offsets are added to I. They are checked to be inbounds only when the pointer is dereferenced. As seen in Section 3.1, this allows free movement of **gep** instructions since they do not depend on the memory state.

[3] Note that memory is deallocated at run time as expected. Also, in our semantics addresses can be reused because allocation functions allocate blocks with addresses that are disjoint from all other *live* blocks only.

4.5 Casting

LLVM has two pointer/integer casting instructions: **ptrtoint** and **inttoptr**. The semantics of casting a logical pointer to an integer is given in PTRTOINT-LOGICAL. Function $\text{cast2int}_M(l, o, s)$ converts $\text{Log}(l, o, s)$ to the integer $P(s)_0 + o$ based on block l. If the operation $P(s)_0 + o$ overflows, it wraps around. This can happen if the pointer is out of bounds. Instruction '**ptrtoint** $\text{Phy}(o, s, I, cid)$' yields o. If the size of the destination type **i**sz is larger than the pointer width, the result is zero-extended. If it is smaller, the most significant bits are truncated.

Casting from an integer to a pointer returns a physical pointer with no provenance information (INTTOPTR). No check is done on whether the pointer refers to a valid location or not. This avoids a dependency on the memory state, and thus allows code motion.

The NULL pointer used in, e.g., C is defined as '**inttoptr** 0' because C programs use '`(void*)0`' as the null pointer value.

Casting a pointer to a different address space through **addrspacecast** translates the pointer's offset(s) using a target-specific mapping function. If the pointer is logical, it preserves the block id and offset if in bounds, otherwise it yields **poison**. If the pointer is physical, '**ptrtoint** $\text{Phy}(o, s, I, cid)$' updates o as well as the offsets in I.

In our semantics, all casting instructions can be freely moved, removed, or introduced.

4.6 Pointer Comparison

To support pointer optimizations, pointer comparison must be defined as something more elaborate than simply comparing the underlying machine addresses. For example, the following program compares two logical pointers that obviously point to different objects. We would like, therefore, to fold this comparison to false:

```
char *p = malloc(4);
char *q = malloc(4);

char *pp = some expr over p;
char *qq = some expr over q;
if (pp == qq) { /* always  false? */ }
```

If `pp` and `qq` are dereferenceable, i.e., their offsets are in the range $[0, 4)$, the comparison should yield `false` since the resulting machine addresses cannot be the same (since objects `p` and `q` cannot overlap). However, what if `pp == p + 4`, `qq == q`, and the objects `p` and `q` were allocated consecutively (i.e., `p + 4 == q`)? Even though `pp` and `qq` are conceptually very different pointers, their underlying machine addresses are the same. Therefore, if the compiler lowers pointer comparison into a comparison of the respective machine addresses in assembly, the comparison would yield `true`.

In our semantics, we define the comparison `p + n == q` to yield a non-deterministic value, justifying both the lowering to machine address comparison, and the desired optimization. This way the compiler can always fold comparisons between different objects to `false` without having to prove that they cannot have the same machine address.

This optimization corresponds to language in the C++ standard that allows an indeterminate result when comparing non-dereferenceable pointers. In contrast, the C standard inconveniently specifies that the physical values of the pointers must be compared. This implies that `p + n == q` must evaluate to `true` if `p` and `q` were allocated consecutively.

Formally speaking, rule ICMP-PTR-LOGICAL' defines that comparing logical pointers to different blocks can always evaluate to `false`. Furthermore, ICMP-PTR-LOGICAL-NONDET-TRUE states that if the offset of either pointer is not dereferenceable, the comparison can *also* evaluate to `true`.

Making pointer comparison non-deterministic violates the C standard. However, both GCC and LLVM (in C mode) will fold a comparison to false even when the pointers compare equal, effectively choosing code quality over standards conformance. The C semantics makes programs harder to optimize since it causes pointer comparisons to leak information about the memory layout, therefore requiring compilers to conservatively assume that most compared pointers escape (which inhibits many optimizations, such as store forwarding and dead store elimination).

A caveat of this choice of semantics is that even if a pointer comparison returns `true`, we cannot assume that two pointers have the same value. However, this was already the case because pointer comparison ignores, e.g., the extra precision fields in physical pointers like *cid*. This makes propagation of pointer equalities (e.g., GVN) unsound. We show later in this section how to make GVN for pointers correct.

For pointer inequality comparison (e.g., `p <= q`), for two logical pointers of the same block, if their offsets are inbounds the result is simply the comparison of their offsets (ICMP-PTR-ULE-LOGICAL). If the offsets are not inbounds, pointer values may overflow in hardware and hence produce a different result than if comparing the offsets. Hence, if one of the offsets is not inbounds, the result of the comparison is nondeterministic, allowing both the compiler to optimize comparisons based on the pointer offsets, as well as efficient compilation of pointer comparisons to assembly.

Comparison of logical pointers into different blocks yields a non-deterministic value. We cannot compare their integer values since that would leak information about the memory layout. We do not make the comparison yield **poison** in this case because optimizations like vectorization introduce comparisons between pointers of potentially different blocks to check at run time if vectorized accesses overlap or not (to check if it is safe to run the vectorized code).

Comparing two physical pointers is equivalent to comparing their integer representations (ICMP-PTR-PHYSICAL, ICMP-ULE-PTR-PHYSICAL). If one pointer is logical and the other physical, the logical pointer is converted to a physical pointer and then they are compared. This naturally supports the definition of comparison with pointer-integer roundtrip in the C/C++ standard: Given a valid pointer `p`, `(void*)(int)p == p` should be true. The pointer-integer round trip yields a physical pointer in our semantics, hence the comparison is always true.

4.7 Pointer Subtraction

Pointers can be subtracted to compute the difference between their offsets. This can be soundly implemented by first casting the pointers to integers and then performing an integer subtraction. In fact, this is what LLVM/Clang does today.

However, there is potential for improvement: The C/C++ standard permits an indeterminate result for the subtraction of pointers into different blocks. However, if the operation was realized as an integer subtraction, such an operation would be well defined and therefore leak information about the memory layout. Therefore, lowering pointer subtraction to subtraction of pointers casted to integers fundamentally loses precision.

In our semantics, if the program subtracts logical pointers from different blocks, the result is a **poison** value. To this end, we introduce a new instruction **psub** that takes two pointers and returns their difference. When given two logical pointers, **psub** gives the difference of their offsets if they refer to the same block (PSUB-LOGICAL). Otherwise, if the pointers refer to different logical blocks, **psub** returns **poison** (PSUB-LOGICAL-POISON). If at least one of the pointers is physical, both pointers are cast to integers and their difference is computed.

Besides the theoretical precision improvement by having a dedicated pointer subtraction instruction, there is also a practical advantage. Compiler analyses are naturally imprecise. In particular,

when they see a pointer-to-integer or an integer-to-pointer cast, they tend to bail out. By introducing a new instruction for pointer subtraction, we are able to reduce the number of these casts significantly (as shown later in the evaluation).

4.8 Memory Block Lifetime

Besides the `p + n == q` case, the machine addresses of different logical pointers may also be equal when addresses get reused by the allocator. For example:

```
char *p = malloc(4);
free(p);
char *q = malloc(4);
if (p == q) { /* ... */ }
```

We would again like to optimize the comparison to `false` because we are comparing the results of two separate calls to `malloc`. And again, in an actual execution, the addresses may be equal because `malloc` can reuse memory after `free`.

A common solution to this problem is to let the behavior of pointer equality depend on whether the block that `p` points to is still allocated. However, the problem with that approach is that it makes comparison not freely reorderable with deallocation. Another solution is to define comparisons with freed blocks as UB (as C and C++ standards do), but that limits movement of pointer comparisons as well. Instead, we add the concept of a *lifetime* to our memory blocks. The memory time stamp gets incremented on every memory allocation and deallocation. The end of the lifetime of a block is initially ∞ and gets set when the block is deallocated.

In the above situation, because the lifetimes of the two blocks do not overlap, we again make pointer comparison non-deterministic, justifying the optimization. This is reflected in ICMP-PTR-LOGICAL-NONDET-FALSE, which also applies if the lifetimes of the blocks are disjoint.

This lifetime-based handling of pointer comparison has the caveat that a `free` can no longer be moved up above a `malloc` of a different block, since that would make the two blocks' lifetimes no longer overlap, possibly affecting program behavior. Although this an interesting optimization to support (to reduce peak memory consumption and potentially reuse cached memory), LLVM does not perform it.

4.9 Load and Store

To perform a memory access of size $sz > 0$ through a pointer p on memory M, the pointer must be *dereferenceable*, written $\mathrm{deref}_M(p, sz)$. If p is a logical pointer $\mathrm{Log}(l, o, s)$ where block l is not freed and $\mathrm{inbounds}_M(l, o) \wedge \mathrm{inbounds}_M(l, o + sz)$, then we have $\mathrm{deref}_M(p, sz)$.

If p is a physical pointer $\mathrm{Phy}(o, s, I, cid)$, there must be a still-alive block $(t, (b, \infty), n, a, c, P)$ with id l and an offset o_l such that $P(s)_0 + o_l = o$ and $\mathrm{inbounds}_M(l, o_l) \wedge \mathrm{inbounds}_M(l, o_l + sz)$. (Note that l and o_l are uniquely determined since memory blocks are disjoint and $sz > 0$.) Moreover, all addresses $o' \in I$ must be inbounds of the same block, i.e., $\forall o' \in I$, $\mathrm{inbounds}_M(l, o' - P(s)_0)$. If the pointer was derived from a parameter (i.e., $cid \neq$ **None**) and the function corresponding to that parameter did not return yet (i.e., $M(cid) \neq$ **None**), then $b < M(cid)$, i.e., the block must have been allocated before the function call identified by cid started. If all these requirements are satisfied, we have $\mathrm{deref}_M(p, sz)$.

To support conversion between values and low-level bitwise representation, we define two meta operations, $ty\Downarrow \in (\llbracket ty \rrbracket \to \mathrm{Bit}^{\mathrm{bitwidth}(ty)})$ and $ty\Uparrow \in (\mathrm{Bit}^{\mathrm{bitwidth}(ty)} \to \llbracket ty \rrbracket)$. For base types, $ty\Downarrow$ transforms **poison** into the bitvector of all **poison** bits, and defined values into their standard low-level representation (Fig. 6a). getbit $v\ i$ is a partial function that returns the ith bit of a value v. If v is a pointer, getbit $v\ i$ returns either **poison** if v is **poison** or a pair (p, i) which is an element

$$\begin{array}{rcl} \mathbf{i}sz\Downarrow(v) \text{ or } ty*\Downarrow(v) & = & \lambda i.\ \text{getbit}\ v\ i \\ \langle sz\times ty\rangle\Downarrow(v) & = & ty\Downarrow(v[0]) +\!\!+ \ldots \\ & & +\!\!+\ ty\Downarrow(v[sz-1]) \end{array}$$

(a) Converting a value to a bit vector

$$\mathbf{i}sz\Uparrow(b) = \begin{cases} n & \text{if } \forall_{0\le i<sz}\ b[i] \neq \textbf{poison} \\ & \text{such that } \forall_{0\le i<sz}\ b[i] = n.i \\ \textbf{poison} & \text{otherwise} \end{cases}$$

(b) Converting a bit vector to an integer

$$\frac{(\iota = \text{“}r = \textbf{load}\ ty, ty*\ op,\ \textbf{align}\ a\text{”}) \quad \text{Load}(M, [\![op]\!]_R, \text{bitwidth}(ty), a)\ \text{fails}}{R, M \overset{\iota}{\hookrightarrow} \text{UB}}$$

$$\frac{(\iota = \text{“}\textbf{store}\ ty\ op_1, ty*\ op,\ \textbf{align}\ a\text{”}) \quad \text{Store}(M, [\![op]\!]_R, ty\Downarrow([\![op_1]\!]_R), a)\ \text{fails}}{R, M \overset{\iota}{\hookrightarrow} \text{UB}}$$

$$\frac{\text{Load}(M, [\![op]\!]_R, \text{bitwidth}(ty)) = v}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto ty\Uparrow(v)], M}$$

$$\frac{\text{Store}(M, [\![op]\!]_R, ty\Downarrow([\![op_1]\!]_R)) = M'}{R, M \overset{\iota}{\hookrightarrow} R, M'}$$

Fig. 6. Semantics of **load** and **store**

of AddrBit denoting the ith bit of a non-poison pointer p. For vector types, $ty\Downarrow$ transforms values element-wise, where $+\!\!+$ denotes the bitvector concatenation.

$\mathbf{i}sz\Uparrow(b)$ transforms a bitwise value b to an integer of type $\mathbf{i}sz$ (Fig. 6b). Notation $n.i$ is used to represent the ith bit of a non-**poison** integer n. Type punning from pointer to integer yields **poison**. This is needed to justify redundant load-store pair elimination.[4] If any bit of b is **poison**, the result of $\mathbf{i}sz\Uparrow(b)$ is **poison**. For vector types, $ty\Uparrow$ transforms bitwise representations element-wise.

$ty*\Uparrow(b)$ transforms a bitvector b into a pointer of type $ty*$. If b is exactly all the bits of a pointer p in the right order, it returns p. Otherwise, it returns **poison**.

Now we define semantics of **load**/**store** operations. Function $\text{Load}(M, p, sz, a)$ returns the bits that correspond to pointer p if $\text{deref}_M(p, sz)$ and if p is a-aligned (i.e., $p \,\%\, a = 0$). **load** yields v if $\text{Load}(M, p, sz, a)$ returns a value v, or UB otherwise. The store operation $\text{Store}(M, p, b, a)$ stores the bit representation b into the memory M and returns the updated memory M' if p is dereferenceable and a-aligned. **store** is UB if $\text{Store}(M, p, b, a)$ fails, and updates the memory to M' otherwise.

4.10 Justifying Transformations in Practice

We now illustrate how our semantics can be used in practice in a compiler, and how one can informally reason about the correctness of optimizations. For example, we would like to justify that the following transformation performed by GVN is correct:

```
char *p = malloc(4);
int v = (int)p;
if (v == 10)
    *(int*)v = 0;
```

$\Rightarrow$

```
char *p = malloc(4);
int v = (int)p;
if (v == 10)
    *(int*)10 = 0;
```

Under our semantics, this transformation is legal because integers do not track provenance. Moreover, the address of object `p` is observed by the comparison, and therefore the store through `(int*)10` is guaranteed to write to object `p`.

A useful way to think about this is that our semantics effectively take *control-flow* dependencies into account when determining which objects have had their addresses observed. In our example, the address of `p` is observed within the then block since it was used in a comparison along the control-flow that leads to the block. Hence, we allow `(int*)10` to refer to object `p`. This sort of reasoning can be directly implemented in a compiler.

[4] Redundant load-store pair elimination means removing ‘`v = load i64 ptr; store v, ptr`’. If reading a logical pointer as integer implicitly casts the pointer, removing this load-store pair would eliminate a cast and hence be illegal. This is discussed further in Section 8.

4.11 Preventing Accesses via Guessed Addresses

We now show how twin allocation semantics prevents unobserved blocks from being accessed via guessed addresses. This aspect is essential to enable compiler optimizations, otherwise any pointer could potentially access any object.

A block is *unobserved* if any value derived from (i.e., any value that has data or control dependence on) the block's logical address created at allocation is never used in any of the *address-observing* operations: being (*i*) casted to an integer, (*ii*) compared to a physical address, and (*iii*) subtracted to/from a physical address. For example, in the following program, the first block of size `10` is unobserved because its address `a` is never used in an address-observing operation, while the second block of size `5` is observed because its address `b` is compared to the physical address `0x200`.

```
1: char *a = malloc(10);
2: char *b = malloc(5);
3: if (b == (char*)0x200)
4:     *(char*)0x100 = 1;
```

A *guessed* address is necessarily a physical address. We will now show that in the twin allocation model, compilers can safely assume that unobserved blocks cannot be accessed via guessed addresses. Since compilers are allowed to assume anything when the source program has an execution raising UB, it suffices to show that whenever an unobserved block b is accessed via a guessed address p in some execution, there is an alternative execution raising UB. The alternative execution is to take the address of the twin block b' instead of that of b at allocation. After allocating blocks b and b', the twin executions (i.e., the original and the alternative) have exactly the same program state except that:

(*i*) the logical address created at allocation corresponds to a different underlying machine address (i.e., that of b and b' respectively), and
(*ii*) validity of b and b' are swapped.

Condition (*i*) does not make any difference in the twin executions because the machine address is only ever used in address-observing operations. More specifically, our semantics is carefully designed in such a way that all operations other than the address-observing ones are independent of the underlying machine address of any logical address. Thus only condition (*ii*) can make a difference during execution. This can happen only when one of the blocks b and b' is accessed via a physical address. Since only one of b and b' is accessible in each of the twin executions, at least one of the executions will raise UB when block b is accessed via a guessed address.

For instance, in the above example, suppose that at line 1, two blocks of size `10` are allocated at `0x100` and `0x150` with the former activated. Then at line 4 the block is successfully accessed via the physical address `0x100`, while in the twin execution where the block at `0x150` is activated, the access at `0x100` at line 4 raises UB. Therefore, the compiler can conclude that the store at line 4 cannot access object `a`.

Finally, we discuss why allocating two blocks is necessary even though one of them is always made invalid. The reason is because in order to have equivalent twin executions modulo accesses via guessed addresses as described above, the twin executions should have identical memory layouts. For example, instead of allocating two blocks, consider allocating a single block in such a way that there is enough space left for allocating another one of the same size; and raising out-of-memory if such allocation is not possible. In this semantics, one may think that we can still simulate a twin execution as described above using the free space. However, it does not work due to different memory layouts. Specifically, in the above example program, suppose that before executing line 1, the memory only has two segments of free space: `0x100` ~ `0x109` and `0x200` ~ `0x209`. Then at line 1, the block of size `10` can be allocated at `0x100` or `0x200` because there is enough space left for

allocating another one. Then at line 2, the block of size 5 can be allocated at `0x200` or `0x205` in the former case and at `0x100` or `0x105` in the latter case because there is enough space left. Among those twin executions, one of them accesses the unobserved block of size 10 via the guessed address `0x100` at line 4 while the others do not reach line 4 thereby raising no UB. Therefore, with this simpler semantics the compiler would (annoyingly) have to conclude that line 4 can access object `a`.

4.12 Sometimes Two Blocks Are Not Enough

Justifying the correctness of some optimizations requires more than two blocks. We give an example of an optimization that requires triple allocation ($N = 2$ in our semantics). This optimization removes single observations of addresses of local variables through comparison with a function argument. For example, in the following function on the left, if the address of `c` is unobserved except at line 3, the comparison can be folded to `false` to get the optimized code on the right. This enables further optimizations since the address of `c` becomes completely unobserved in the target code and thus the compiler can assume that the block `c` cannot be accessed via a guessed address.

```
1: void foo(int i) {                      void foo(int i) {
2:   char c[4];                             char c[4];
3:   if (c == (char*)i) {                   if (false) {
4:     ...                                    ...
5:   } else {                      ⇒         } else {
6:     *(char*)0x200 = 0;                     *(char*)0x200 = 0;
7:   }                                       }
8:   ...                                     ...
9: }                                       }
```

The reason why two blocks are not enough for this optimization is that `c` can be accessed in line 6 of the original code when memory has only space for two blocks: The observation of the address of `c` in line 3 can forbid one of the twin executions from reaching line 6, which does not happen in the optimized code. To see this clearly, suppose that `i = 0x100` and the twin blocks for `c` can be only allocated at `0x100` and `0x200` because the memory had space for just those blocks. Then, in the original code, none of the twin executions triggers UB for the guessed access at line 6, while in the optimized code, the execution with `c = 0x100` triggers UB at line 6 now that guessing is prevented, as usual, by the twin allocations.

However, if we have triple twin allocations, `c` cannot be accessed via a guessed address even if one of the triple twin executions is ruled out by a single observation: We still have two twin executions left. For example, in the above setting, if `c` is allocated at `0x100`, `0x200` and `0x300`, in the original code the execution with `c = 0x300` triggers UB for the guessed access at line 6.

We can argue as follows that triple twin allocations enable the optimization above for a fresh block `c` when the address of `c` is observed only once in the comparison. If one of the triple executions in the optimized code accesses `c` via a guessed address (causing UB in the optimized code), we can always trigger UB for the guessed access in one of the three possible executions of the original code as discussed above. If not, the triple executions behave exactly the same in the optimized code because there is no observation of the address of `c`, and the behavior is simulated by one of the triple executions of the original code that makes the comparison `c == p` false.

Finally, we note that over-approximating the number of twin blocks is sound. However, as we have seen in the previous example, under-approximating it is not. In practice, due to the limited reasoning power of compilers in general, and LLVM in particular, we believe 3 blocks are sufficient. [5]

[5] We are only aware of one optimization in LLVM and GCC that may require more than three blocks, and further investigation is ongoing to determine whether it is valid (see http://llvm.org/PR35102).

5 IMPLEMENTATION

We implemented two prototype versions of LLVM to study the impact of adapting the compiler to match the new semantics. First, we removed optimizations that are invalid in our semantics in order to create a version of LLVM that soundly implements its memory model. We confirmed this version fixes all the related bugs we have found. Second, we modified the "sound" compiler to regain some performance by adding optimizations that were not previously supported. We implemented our prototypes using LLVM 6.0.[6]

Making LLVM sound. LLVM propagated pointer equalities in several places; these had to be disabled. For example, InstSimplify (a peephole optimizer that does not create new instructions) transforms the IR equivalent of `(x == y) ? x : y` into `y`. In our semantics this transformation is correct for integers, but not for pointers, because it breaks data-flow provenance tracking. Similarly, we turned off GVN for pointer-typed variables.

A second kind of transformation that we had to disable were some integer-pointer conversions. LLVM (and GCC to some extent) treat a round-trip of pointer-to-integer and then integer-to-pointer casts as a no-op. This is incorrect under our semantics, and we removed this transformation—the IR equivalent of turning `(int*)(int)p` into `p`—in InstSimplify. Also related is a transformation from InstCombine (a peephole optimizer that potentially creates new instructions), which rewrites `(int)p == (int)q` into `p == q` for pointers `p` and `q`. This transformation was also removed, since comparing physical pointers is not equivalent to comparing logical pointers in our semantics.

A third kind of change is related to compiler-introduced type punning. There are optimizations in LLVM that transform, for example, load/store instructions of pointers into that of pointer-sized integers. This includes GVN when equivalent variables have different types, transforming a small `memcpy` into a pair of load/store instructions (e.g., where the intermediate value is an integer, and the target type is a pointer), etc. As we will discuss in Section 8, we had to disable all of these.

Finally, we disabled a few additional transformations that became invalid in our semantics including folding a pointer comparison against a stack-allocated object within a loop and a handful of peephole optimizations that changed pointer provenance. In total, we changed 419 lines of code.

Regaining performance. Making LLVM sound with respect to our memory model caused some small performance regressions in SPEC CPU 2017. We created a second prototype that attempts to regain that performance by adding optimizations that conform to our new semantics.

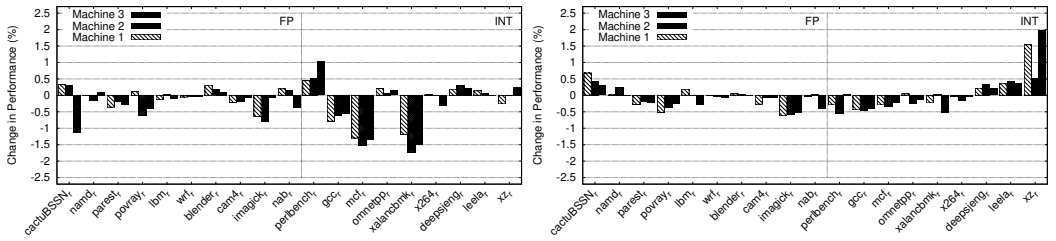
We changed how LLVM/Clang handles C/C++ pointer subtractions. The unmodified Clang lowers pointer subtraction into the IR equivalent of `(int)p - (int)q`. This is sound under our model, but since LLVM is very conservative when it encounters these kind of casts (it assumes the pointers escape), we unnecessarily lose precision in, e.g., alias analysis. The problem becomes even worse in our "sound" prototype. We added the **psub** instruction for pointer subtraction (described in Sec. 4.7) and modified Clang to use it. Since in our semantics **psub** does not escape if both operands are logical pointers, more aggressive optimization are now allowed.

We augmented alias analysis so that it becomes more precise when it encounters integer-to-pointer casts. Alias analysis can now conclude that a pointer cast from integer never aliases an unescaped object. Moreover, we also improved the analysis to assume that the pointer comparison instruction does not make its operands escape if they are both logical pointers.

Finally, we re-enabled GVN for pointer types for the following specific cases where it is guaranteed to be sound (for pointers `p` and `q` in the same equivalence class, where `p` is replaced with `q`):

- `q` is `NULL` or the result of an integer-to-pointer cast.
- `p` and `q` are logical pointers, and both are either dereferenceable or point to the same block.

[6] https://github.com/snu-sf/llvm-twin and https://github.com/snu-sf/clang-twin



(a) After making LLVM sound (b) Sound prototype + optimizations

Fig. 7. Change in run time of SPEC CPU 2017 for our two prototypes. Higher bars indicate speedup relative to baseline LLVM, lower bars indicate slowdown.

- p and q are both computed by the **gep inbounds** with same base pointer.
- Either p or q is computed by a series of **gep inbounds** with positive offsets, based on the same base pointer.

The number of lines changed in this prototype against the previous one totals about 1274 lines. Overall, our "sound and fast" prototype requires changing less than 1700 lines of code in LLVM to make it sound against our memory model with minor impact on run time performance.

6 PERFORMANCE EVALUATION

For our memory model to have a chance of being adopted, we must establish that it does not interfere with the quality of generated code.

6.1 Setup

We used three machines, with the following CPUs, to measure performance: (1) Intel i3-6100 (Skylake), (2) Intel i5-6600 (Skylake), and (3) Intel i7-7700 (Kaby Lake). All machines have 8 GB RAM and Ubuntu 16.04 installed. To get more consistent results, Intel Hyperthreading, TurboBoost, SpeedStep, and Speed Shift were disabled, and the CPU scaling governor was set to "performance." We also disabled non-essential system services and address space layout randomization.

As benchmarks, we used SPEC CPU 2017 and the LLVM Nightly Test suite (293 C/C++ benchmarks). For SPEC CPU, we used only the SPECrate benchmarks because of RAM limitations (SPECspeed requires more memory than our machines had). The Fortran benchmarks were compiled first with `gfortran`.

Each benchmark was compiled with `-O3`, executed three times, and the median running time is used to compare performance. To run LLVM Nightly Tests, we used the `cpuset` utility to pin the benchmark to a single core. Unlike SPEC, these short-running benchmarks were seeing significant variance due to migrations between cores. Also, we used a RAM disk to minimize fluctuations due to disk access time. Some benchmarks in the LLVM suite have short running time, so using this setting is helpful to stabilize the results.

When comparing performance for LLVM Nightly Tests, we exclude benchmarks where our changes to Clang and LLVM did not result in any changes to the resulting object code. Some benchmarks had high performance variation (> 5%) across runs, most of these were benchmarks that run too quickly for accurate timing. We excluded these from our results as well.

6.2 Performance of Generated Code

Performance after making LLVM sound. Figure 7a shows the effect that our first (sound but unoptimized) prototype had on the performance of SPEC CPU 2017. On average, across all benchmarks and machines, we observed a 0.2% slowdown. The worst slowdown was a 1.75% regression

in xalancbmk on machine 2. There was also a 0.5 ∼ 1.5% consistent slowdown in gcc, mcf, and xalancbmk.

For the LLVM Nightly Tests, the worst slowdown was 4.3%, and the best speedup was 4.0%. On average, we observed a 0.3% slowdown. One benchmark, Oscar, had a huge speedup (25%). This was because our fixes prevented loop vectorization from working in this case, which happened to make the benchmark run faster. This benchmark is omitted in the numbers mentioned before.

In the LLVM Nightly Tests, only 25% (464/1853) of object files were different than when compiled with the baseline compiler, and only 27% (80/293)of benchmarks had any object file changed.

Across all of our benchmarks, the baseline version of LLVM's GVN pass performs about 24,700 replacements. About 28% of these correspond to propagation of pointer equalities, with the remaining being propagation of integer equalities. Of the pointer replacements, 44% correspond to replacing a pointer with `NULL`, which is a sound transformation under our semantics (even though it was disabled in this prototype).

Performance after adding optimizations. Fig. 7b shows the change in performance of SPEC CPU 2017 for our second prototype ("sound and fast") relative to baseline LLVM. The three changes that contributed the most to recovering lost performance were as follows. Adding **psub** changed the average slowdown on SPEC from −0.2% to −0.1%. The main winners due to this optimization were mcf and xalancbmk, having a 1.7% and 1.1% speedup, respectively.

We found "jpeg-6.a" in LLVM Nightly Tests showed 2.2% slowdown. It was because removing transformations from load/store of pointers into load/store of pointer-sized integers blocked SLP vectorization of heterogeneous types. We can support this by allowing type punning between a physical pointer and integer. Formally, we can define load of integer i as pointer to yield $\mathrm{Phy}(i, 0, \emptyset, \textbf{None})$ rather than **poison**, and load of $\mathrm{Phy}(i, 0, \emptyset, \textbf{None})$ as integer to return i. With this update, it is valid to canonicalize load/store of pointer-sized integers into load/store of pointers. After the canonicalization, SLP vectorization can fire because load/stores now have homogeneous types. We implemented this canonicalization pass before SLP vectorization and checked that the slowdown disappeared. It did not, however, improve the performance of the SPEC benchmarks nor the average slowdown of the LLVM Nightly Tests. This canonicalization can be a good workaround for reenabling vectorization, but in the future we would like to pursue more generalized solution for this, as discussed in Section 8.

Finally, reactivating GVN for safe cases removed the slowdown in perlbench and cactuBSSN. With the Nightly Tests, the performance regressed slightly, increasing the average slowdown to 0.07%.

This experiment shows that it is possible to make LLVM adhere to our memory model without significant performance regression. In fact, some benchmarks get a small performance improvement.

6.3 Number of Instructions

We investigated how the number of pointer cast instructions (**inttoptr** / **ptrtoint**) and **psub** instructions evolved with our modifications. Table 1 shows the total number of instructions over all the benchmarks, when compiled without optimizations (`-O0`) and with optimizations (`-O3`).

In the first column, we have the numbers for the baseline (vanilla LLVM). In the second column, we can observe the number of casts does not increase significantly after disabling the (incorrect) optimization that removes the round-trip cast of `(int*)(int)p` to `p`, for a pointer `p`.

The third column shows the result for our first prototype, where all incorrect optimizations were removed. The number of **inttoptr** instructions is reduced because we disabled the introduction of type punning by the optimizer. The increase in the number of **ptrtoint** casts is the result of removing unsound optimizations like the one that rewrites `(int)p == (int)q` into `p == q`.

		Baseline	**i2p(p2i** p**)** fold disabled	Sound	add **psub**
-O0	Total	21,219,982	21,219,982	21,219,982	21,194,610
	inttoptr	1,554	1,554	1,554	1,554
	ptrtoint	39,726	39,726	39,726	14,338
	psub	0	0	0	12,806
-O3	Total	15,015,064	15,030,020	14,950,405	14,750,101
	inttoptr	29,976	44,255	2,580	2,573
	ptrtoint	38,249	45,425	64,585	9,431
	psub	0	0	0	29,117

Table 1. Number of instructions with different prototypes

In the last column, we show the data for a prototype that uses the new **psub** instruction. Not only are the number of casts reduced in the non-optimized build (since we changed Clang to lower pointer subtraction into **psub**), but the optimized build also shows a reduction since we also added support for the optimizers to create **psub** when needed.

Finally, we note the number of cast instructions is still larger in the optimized build since optimizations like loop unrolling and inlining can soundly duplicate instructions.

6.4 Compile time

We measured the time it takes to compile each file of all the benchmarks in an optimized build. We exclude files with a very small compile time (< 0.15 s) in these results.

In the first prototype, the average compile time had a 0.1% slowdown. The worst slowdown was 10.5% and the best speedup 13.9%. For the second prototype, we observed an average slowdown of compilation time of 1.1%. The worst slowdown increased to 9.3%. A reason for the second prototype regressing further on the compilation time is due to extra analysis required to propagate pointer equalities. Also, our prototype implementations were not optimized to reduce compilation time.

7 RELATED WORK

Much work has been done on defining and clarifying the semantics of C, including the semantics of pointer provenance [Chisnall et al. 2016; Hathhorn et al. 2015; Krebbers 2013; Krebbers and Wiedijk 2015; Memarian et al. 2016; Memarian and Sewell 2016a,b]. However, focusing on source language semantics involves different considerations than that of a compiler IR. For example, many of the optimizations presented in this paper are not sound under the proposed C semantics. That is not a problem, as long as C can efficiently be compiled to an IR like LLVM's that allows the desired optimizations.

CompCert [Leroy 2009; Leroy and Blazy 2008] is a compiler for a subset of C that is formalized in Coq. It defines a pointer as a pair of block id and offset, which is equivalent to our logical pointer. CompCert does not, however, support pointer-to-integer casts, thereby avoiding the problems we address in this paper. Ševčík et al. [2013] extend CompCert with a weak memory model.

Vellvm [Zhao et al. 2012] formalizes the LLVM IR in Coq. It adopts CompCert's memory model, inheriting its weaknesses.

Besson et al. [2014; 2015; 2017a; 2017b] propose an extension of the CompCert memory model supporting casts between integers and pointers. When a pointer is cast to an integer, a fresh symbol is created to represent the underlying physical address and henceforth keep the result of arithmetic or bitwise operations over it as symbolic expressions. When a concrete value is needed for a symbolic expression, if it uniquely determines an integer value, then that value is returned; otherwise, poison is returned. They verify CompCert optimizations and programs performing bitwise operations on pointers with their memory model. However, their model is limited in practice

	high-level mem optimizations	int-to-ptr casts	low-level ptr manipulation	all integer optimizations	ptr arith is pure
Flat		✓	✓	✓	✓
Memarian et al.	✓	✓			✓
CompCert	✓			✓	✓
Besson et al.	✓	✓			✓
Kang et al.	✓	†	✓	✓	
Ours	✓	✓	✓	✓	✓

Table 2. Comparison between different memory models: whether they enable high-level memory optimizations (such as store forwarding), whether they support integer-to-pointer casts, whether they support languages with low-level pointer manipulation (such as doing XORs of pointers), whether they support all standard integer optimizations (such as equality propagation and reassociation), and whether pointer arithmetic and cast operations are pure (since otherwise they constrain movement of such instructions). † does not support cast of $p + n$.

because observing the underlying address of a pointer is not allowed. For example, using a pointer value as a hash key is undefined in their model.

Kang et al. [2015] proposed a low-level memory model that supports pointer-to-integer casts. However, the model does not support casting a pointer one-past-the-end of an object to an integer and back: this round-trip resulted in **poison** in their semantics, while it is well-defined in C. Moreover, in their semantics, `ptrtoint` is an effectful operation that cannot be removed even if the result is not used, and `inttoptr` is not freely movable across `malloc` and `free`. Our work fixes both of these problems by combining logical and physical pointers in one memory model.

Table 2 shows a summary comparing the different memory models and ours.

Lee et al. [2017] proposed adjusting the semantics of **poison** and **undef** in order to fix some problems in LLVM. They do not address problems with pointer values and thus their work is orthogonal to ours, and in fact we reuse their value semantics for our work.

Both Alive [Lopes et al. 2015] (an automatic peephole optimization verifier) and Souper [Sasnauskas et al. 2017] (a superoptimizer) encode the semantics of LLVM IR into SMT. Souper currently does not support memory operations, and Alive does not support integer-to-pointer casts. Both could be extended with our new memory model in order to support a larger set of transformations.

Verification tools for C/C++ programs often use memory models that try to split objects into different heaps to speedup verification time (e.g., [Gurfinkel and Navas 2017; Wang et al. 2017]). However, usually these models either do not support integer-to-pointer casts or they fallback to a flat memory model when these casts are used. The trade-offs for IR and verification tools memory models are different, since verification tools can soundly lose precision for uncommon cases, for example, while for compiler IRs it cannot since the IR memory model needs to justify the correctness of transformations in all cases. SMT-based verification have highly-optimized SMT encodings of their memory models, while ours is not trivial to encode efficiently.

8 DISCUSSION AND FUTURE WORK

Type Punning. In our semantics, loading, e.g., from a memory location that contains a pointer at an integer type results in **poison** due to type punning. One may wonder why we did not specify the load to, instead, perform an implicit pointer-to-integer cast. Disallowing implicit pointer casts is necessary to justify load-store elimination:

```
int q, **x = malloc(sizeof(int*));
*x = &q;
int i = *(int*)x // implicit pointer-to-integer cast
*(int*)x = i; // remove; it is redundant
```

So, while the original code performed a pointer-to-integer cast and then wrote back an integer (or, equivalently, a physical pointer with no provenance information), the changed program leaves the original provenance information intact. In other words, load-store elimination can result in increasing provenance restrictions on pointers, which can in turn result in introducing UB. To fix this, we let load return **poison** instead of performing an implicit cast.

The downside of disallowing implicit casts is that there is no longer a type that can actually hold any data that can be present in memory. LLVM typically uses integer types for that purpose but, as discussed above, that does not work in our semantics. One option to fix this is adding a new "data" type that only permits a few bitwise operations, and that can hence hold both pointer and integer bits. We intend to pursue this avenue further in the future.

Non-deterministic Values. We have defined some operations to yield a non-deterministic result, e.g., **icmp ule** of pointers of different blocks. This blocks duplication of instructions since they could then return different results. Solutions to this problem could include use of **undef** instead, which has its drawbacks [Lee et al. 2017], or explore the use of "entangled" instructions that need to be folded together. We leave this exploration for future work.

Out of memory. A technical problem with our memory model is that any transformation that may eliminate an out-of-memory condition is unsound. However, the necessary conditions for achieving miscompilation via this unsoundness are difficult to meet. First, the compiler must perform a transformation based on knowledge that a program path triggers out-of-memory (LLVM does not do this at present, and seems unlikely to do so in the future). Second, it would need to perform an optimization eliminating the out-of-memory condition (these transformations, such as eliminating unused allocations, are performed routinely). Designing a memory model that is sound in this respect seems challenging and it is not clear the result would be useful in practice.

9 CONCLUSION

Languages like C, C++, and Rust give the programmer low-level control over how memory is arranged and accessed while also giving the compiler freedom to perform high-level memory optimizations. This is challenging and current toolchains based on LLVM and GCC are not always able to strike the right balance between performance of generated code and strength of guarantees made to programs. Part of the problem is that the memory models for the internal representations in these compilers—where high-level optimizations are performed—are informally specified and thus easily misunderstood.

We created a new memory model for LLVM IR that we believe provides sufficiently strong guarantees that it can be efficiently targeted by front-ends while also permitting many desirable optimizations. We have implemented a prototype of this model and have shown that the implementation is not invasive, that code quality remains good, and that several known miscompilations due to the current, informal memory model are fixed.

ACKNOWLEDGMENTS

The authors thank Sanjoy Das, Alberto Magni, and Matthew Parkinson for feedback and discussions on previous versions of this paper. This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (2017R1A2B2007512), by an ERC Consolidator Grant for the project "RustBelt" (grant no. 683289), and by the National Science Foundation (grant no. 1218022). The first author was supported by a Korea Foundation for Advanced Studies scholarship.

A APPENDIX: END-TO-END MISCOMPILATION BY BOTH LLVM AND GCC

This C program is miscompiled by both GCC and LLVM:[7]

```
$ cat b.c
void f(int *x, int *y) {}

$ cat a.c
#include <stdint.h>
#include <stdio.h>

void f(int *, int *);

int main(void) {
  int a = 0, y[1], x = 0;
  uintptr_t pi = (uintptr_t)&x;
  uintptr_t yi = (uintptr_t)(y + 1);
  int n = pi != yi;

  if (n) {
    a = 100;
    pi = yi;
  }

  if (n) {
    a = 100;
    pi = (uintptr_t)y;
  }

  *(int *)pi = 15;

  printf("a=%d x=%d\n", a, x);

  f(&x, y);

  return 0;
}

$ clang-5.0 -Wall -O2 a.c b.c ; ./a.out
a=0 x=0
$ gcc-7 -Wall -O2 a.c b.c ; ./a.out
a=0 x=0
```

The result produced by both compilers is incorrect. There are only two possible outcomes, depending on whether x and y are allocated consecutively (n is false) or not (n is true):

- Case 1: n is true and the program must print
  `a=100 x=0`
- Case 2: n is false and the program must print
  `a=0 x=15`

[7]Please note that the program may produce different results on different platforms, since the miscompilation is only observable with some stack layouts. Changing the order of declaration of variables x and y is often sufficient to observe the miscompilation.

No other output is permitted. In both compilers, the root cause is insufficiently conservative treatment of a pointer derived from an integer. The problem is difficult to solve without throwing away desirable pointer optimizations. This program has been reported in the bug tracking systems for both compilers. The semantics given in this paper solve this problem, and we have confirmed that our prototype fixes this bug.

B APPENDIX: SAFE RUST PROGRAM MISCOMPILED BY LLVM

This function uses low-level language features, but it is still in the safe subset of Rust. It is miscompiled because of a bug in LLVM's GVN optimization:

```
pub fn test(gp1: &mut usize, gp2: &mut usize, b1: bool, b2: bool) -> (i32, i32) {
    let mut g = 0;
    let mut c = 0;
    let y = 0;
    let mut x = 7777;
    let mut p = &mut g as *const _;

    {
        let mut q = &mut g;
        let mut r = &mut 8888;

        if b1 {
            p = (&y as *const _).wrapping_offset(1);
        }
        if b2 {
            q = &mut x;
        }

        *gp1 = p as usize + 1234;
        if q as *const _ == p {
            c = 1;
            *gp2 = (q as *const _) as usize + 1234;
            r = q;
        }
        *r = 42;
    }
    return (c, x);
}
```

This function first assigns a reference of `g` to `q`. It then creates a temporary object holding the number `8888` and `r` is assigned a reference to it. Function `wrapping_offset` performs pointer arithmetic that can safely go out of bounds of the base object (equivalent to LLVM's **gep** without **inbounds**). If `b2` is true, `q` is assigned a reference to `x`. Therefore, if the program enters in the following branch as well, `r` is assigned a reference to `x`, and thus the following store through `r` (`*r = 42;`) overwrites the value of `x`.

When called with `b1` and `b2` set to true, the optimized version of this code returns `c = 1, x = 7777`. This outcome is impossible; legal results are `c = 0, x = 7777` and `c = 1, x = 42`.

The miscompilation happens when LLVM's GVN pass exploits the condition of the last `if` statement and incorrectly replaces all uses of `q` within that branch with `p`. After the replacement, `*r` is assumed not to touch `x` because `r` now either contains a reference to the initial temporary object or to one of the references assigned to `p` (based on `g` and `y` only). Therefore, `x = 7777` is constant-propagated to the return statement, which causes the wrong output.

C APPENDIX: COQ FORMALIZATION AND PROOF

We formalized our memory model in Coq and proved several key claims of this paper. The code is available from https://github.com/snu-sf/llvmtwin-coq. Note that we omit function calls and returns to simplify the formalization.

C.1 Definitions

The memory model is specified in file `Memory.v`. It differs in two ways from the presentation in this paper: (1) it does not support address spaces for brevity, and (2) memory maintains the last used block id, which is used to create fresh ids on allocation. The number of twin blocks ($|P|$) is set to 3.

Well-formedness of a memory block is defined in `Ir.MemBlock.wf`, and states, e.g., that $|P|$ is always 3, and the size of blocks is larger than zero. Well-formedness of memory is defined in `Ir.Memory.wf` and states, e.g., that all existing memory blocks are well-formed, and alive blocks have no overlapping addresses. Well-formedness of a state is defined in `Ir.Config.wf`, and includes assertions like the program counter is valid, memory is well-formed, etc. We proved that the execution of any instruction preserves the well-formedness of the input state.

The small-step semantics is defined in file`SmallStep.v`. Given an input state, a step can result in one of the following results: (1) success, if the execution of the following instruction was successful and yielded a new state, (2) UB, if the program raised undefined behavior, (3) OOM, if the program raised out-of-memory, and (4) halt, if the program finished.

Two states s and s' are twin with respect to a memory block l iff they are equal except for the configuration of l where the addresses of l in s' (P') are a permutation of those of s (P) and the enabled address is different (i.e., $P_0 \neq P'_0$).

A block's address is observed if a pointer to that block is given as argument to one of the following instructions: **ptrtoint**, **psub**, or **icmp**. Moreover, for the latter two instructions, the other argument must be a physical pointer. A pointer is guessed if it points to an unobserved block.

C.2 Proofs

We proved the following theorems in Coq:

Theorem C.1 (Twin allocation forbids pointer guessing). *Given two twin states s and s' w.r.t. l, where the next instruction dereferences a guessed pointer to l, the execution of either s or s' triggers UB.*

Two additional theorems are proved that establish the usefulness of twin states:

Theorem C.2 (Malloc). **malloc** *either returns a NULL pointer, or a logical pointer* $\mathsf{Log}(l, o)$. *Moreover, the states yielded by the small-step execution are all twin w.r.t. l.*

Theorem C.3. *Given two twin states s and s' w.r.t. l, the small-step execution of s and s' where the next instruction does not dereference a guessed pointer and does not observe block l either (1) halts, triggers UB or OOM, or (2) is successful and the successor states of s are twin with those of s'.*

We now show that certain instructions can be freely moved across (de)allocation functions:

Theorem C.4 (Instruction reordering). *Instructions* **icmp eq**, **icmp ule**, **psub**, **inttoptr**, **ptrtoint**, *and* **gep** *can be moved across* **malloc** *and* **free** *in both directions (upward and downward). Moreover, a program P' obtained from P by doing such reordering is equivalent to P.*

Finally we prove sufficient conditions for soundness of GVN:

Theorem C.5 (Soundness of GVN for pointers). *The four conditions given in Section 5 that state when it is sound to replace a pointer* `p` *with another pointer* `q` *are correct, i.e., the pointer replacement is a refinement.*

REFERENCES

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2014. A Precise and Abstract Memory Model for C Using Symbolic Values. In *APLAS*. https://doi.org/10.1007/978-3-319-12736-1_24

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2015. A Concrete Memory Model for CompCert. In *ITP*. https://doi.org/10.1007/978-3-319-22102-1_5

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2017a. CompCertS: A Memory-Aware Verified C Compiler Using Pointer as Integer Semantics. In *ITP*. https://doi.org/10.1007/978-3-319-66107-0_6

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2017b. A Verified CompCert Front-End for a Memory Model Supporting Pointer Arithmetic and Uninitialised Data. *Journal of Automated Reasoning* (03 Nov 2017). https://doi.org/10.1007/s10817-017-9439-z

David Chisnall, Justus Matthiesen, Kayvan Memarian, Peter Sewell, and Robert N. M. Watson. 2016. C memory object and value semantics: the space of de facto and ISO standards. https://www.cl.cam.ac.uk/~pes20/cerberus/notes30.pdf

Arie Gurfinkel and Jorge A. Navas. 2017. A Context-Sensitive Memory Model for Verification of C/C++ Programs. In *SAS*. https://doi.org/10.1007/978-3-319-66706-5_8

Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *PLDI*. https://doi.org/10.1145/2737924.2737979

Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A Formal C Memory Model Supporting Integer-pointer Casts. In *PLDI*. https://doi.org/10.1145/2737924.2738005

Robbert Krebbers. 2013. Aliasing Restrictions of C11 Formalized in Coq. In *CPP*. https://doi.org/10.1007/978-3-319-03545-1_4

Robbert Krebbers and Freek Wiedijk. 2015. A Typed C11 Semantics for Interactive Theorem Proving. In *CPP*. 15–27. https://doi.org/10.1145/2676724.2693571

Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *PLDI*. https://doi.org/10.1145/3062341.3062343

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *Journal of Automated Reasoning* 41, 1 (Jul 2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *PLDI*. https://doi.org/10.1145/2737924.2737965

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N.M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. In *PLDI*. https://doi.org/10.1145/2908080.2908081

Kayvan Memarian and Peter Sewell. 2016a. Clarifying the C memory object model (revised version of WG14 N2012). https://www.cl.cam.ac.uk/~pes20/cerberus/notes64-wg14.html

Kayvan Memarian and Peter Sewell. 2016b. N2090: Clarifying Pointer Provenance (Draft Defect Report or Proposal for C2x). https://www.cl.cam.ac.uk/~pes20/cerberus/n2090.html

The LLVM Project. 2018. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *CoRR* abs/1711.04422 (2017). http://arxiv.org/abs/1711.04422

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (June 2013), 22:1–22:50 pages. https://doi.org/10.1145/2487241.2487248

Wei Wang, Clark Barrett, and Thomas Wies. 2017. Partitioned Memory Models for Program Analysis. In *VMCAI*. https://doi.org/10.1007/978-3-319-52234-0_29

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *POPL*. https://doi.org/10.1145/2103656.2103709