
# OOElala: Order-of-Evaluation Based Alias Analysis for Compiler Optimization

Ankush Phulia*
Indian Institute of Technology Delhi
India

Vaibhav Bhagee*
Indian Institute of Technology Delhi
India

Sorav Bansal
Indian Institute of Technology Delhi
India

## Abstract

In C, the order of evaluation of expressions is unspecified; further for expressions that do not involve function calls, C semantics ensure that there cannot be a data race between two evaluations that can proceed in either order (or concurrently). We explore the optimization opportunity enabled by these non-deterministic expression evaluation semantics in C, and provide a sound compile-time alias analysis to realize the same. Our algorithm is implemented as a part of the Clang/LLVM infrastructure, in a tool called *OOElala*. Our experimental results demonstrate that the untapped optimization opportunity is significant: code patterns that enable such optimizations are common; the enabled transformations can range from vectorization to improved instruction selection and register allocation; and the resulting speedups can be as high as 2.6x on already-optimized code.

***CCS Concepts:*** • **Software and its engineering → Concurrent programming structures**; **Automated static analysis**; **Software performance**.

***Keywords:*** Undefined Behaviour, Compiler Optimization

**ACM Reference Format:**
Ankush Phulia, Vaibhav Bhagee, and Sorav Bansal. 2020. OOElala: Order-of-Evaluation Based Alias Analysis for Compiler Optimization. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3385962

## 1 Introduction

A programming language like C has several types of non-determinism, categorized as "unspecified behaviour", "implementation-defined behaviour", and "undefined behaviour". Historically, such non-deterministic language features have ostensibly been introduced for "portability"; however, a closer look often reveals that these non-deterministic features are sometimes performance choices, or "performance portability" choices. An example of such a non-deterministic feature is C's left-shift operation $a << b$, which has undefined behaviour (UB) if $b$'s value is greater than $a$'s bitwidth. Leaving the behaviour undefined in this situation allows efficient lowering of the left-shift operator on multiple ISAs, each of which may have differing (well-defined or undefined) semantics for the same situation.

When compared with other programming languages, an optimization-centric language like C has perhaps retained the most amount of non-determinism in its abstractions. A well-known non-deterministic feature of C is the *unspecified Order- Of-Evaluation (OOE) for unsequenced computation and its associated UB*. It is worth considering the foundational reasons behind leaving the order-of-evaluation for C expressions unspecified: perhaps the intent was to (a) allow greater flexibility in instruction selection and scheduling while implementing expression evaluation on single-threaded CPUs, e.g., instructions can now be interleaved, reordered, or vectorized; and (b) allow efficient implementations of expression evaluation for parallel architectures like FPGAs and GPUs (e.g., evaluate unsequenced expressions in parallel).

Even though this non-determinism in expression evaluation has existed since the early days of C89, all popular compilers "determinize" it by picking a fixed pre-determined OOE at a very early stage in the compilation pipeline. Further, this determinization is different for different compilers. All compiler analyses and optimizations are then performed on this deterministic version of the program. As a consequence, these compilers ignore the UB associated with potential data races in unsequenced expression evaluation. These compiler decisions also reflect in the evolution of the programming language (PL) over time. For example, in PL design, there has been a general trend to avoid non-determinism as it is seen to only abet programming errors.

*Contributions*: we are interested in the following four questions: (1) What are some additional optimization opportunities that C's OOE non-determinism enables for a single-threaded CPU? We do not identify all such opportunities, but cover a significant set. (2) What is the compiler support required to exploit the additional optimization opportunity, and what are the resulting improvements? (3) How commonly are such coding patterns encountered in real-world

*Both authors contributed equally to the paper



*PLDI '20, June 15–20, 2020, London, UK*

ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3385962

code, where these improvements are possible? (4) Is the programmer coding such patterns "consciously"?

We focus on the UB associated with data races that may occur during the evaluation of the unsequenced portions of a C expression. We refer to such data races as *unsequenced races* in the rest of this paper, to distinguish them from the data races caused due to multi-threading. We describe a static analysis pass on the program's abstract syntax tree (AST) that infers must-not-alias relationships between the program values based on these UB semantics, and makes the results of this analysis available to subsequent transformation passes. We have implemented our algorithm inside the Clang/LLVM framework, and refer to the enhanced compiler as OOElala[1]. We find that the extra must-not-alias information generated by our tool enables optimizations like register-allocation and vectorization, leading to significant (up to 2.6x) performance improvements in some cases. *We find that such coding patterns, that allow OOE non-determinism to be leveraged for optimization, are fairly common in real-world code* — our experiments, on programs drawn from the SPEC CPU 2017 benchmark suite, show that the extra information has an effect on the compiler transformations performed in 165 of the 772 C (source code) files belonging to these programs.

*Further we derive an LLVM-based sanitizer from our static analysis algorithm to catch OOE-based UB at runtime.* We exercised our sanitizer by running the benchmarks compiled with OOElala with the sanitizer's runtime checks, and found zero assertion failures on the reference tests available with the SPEC CPU 2017 benchmarks. This indicates that the programmers' code patterns were perhaps not bugs, but conscious choices. As such, we think that our work presents an interesting case-study on the uses of non-determinism. While it is true that non-determinism encourages programming errors, disabling it may not always be the best solution: sometimes, it may be more appropriate to provide the required tool support (e.g., UB sanitizer) to detect errors early.

This paper shouldn't be seen as favoring the inclusion of more non-determinism in programming languages. Our interest lies in evaluating the merits of non-determinism to provide more context to the larger discussion on PL design.

Before describing the relevant C semantics and our algorithm, we motivate the discussion through two examples:

```
// compute the indices of
// min/max elements in 'a'
int n;
void minmax(double *a,
    int *min, int *max) {
*min = *max = 0;
for(int i=0; i<n; i++) {
 *min=(a[i]<*min)?i:*min;
 *max=(a[i]>*max)?i:*max;
}
}
//Improvement: 50%
```

```
//538.imagick_r
//morphology.c line 2083
for (i=0,v=-kernel->y;
    v<=kernel->y; v++)
 for (u=-kernel->x;
    u<=kernel->x; u++,i++)
  kernel->positive_range +=
    (kernel->values[i] =
      args->sigma*MagickMax(
        fabs((double)u),
          fabs((double)v)));
//Improvement: 66% (2 calls)
```

[1]OOElala is derived from "Order-Of-EvaLuation based ALias Analysis".

The first example (left) computes the indices of the minimum and maximum elements in an array; the second example (right) is taken from SPEC CPU 2017 benchmarks' `imagick` application and involves initialization of a "kernel" matrix for an image processing application. In both examples, the expression that exhibits OOE non-determinism is underlined, and because these expressions involve multiple memory accesses (of which at least one is a write access), we are able to infer that for these expressions to have well-defined behaviour on any initial state, certain must-not-alias relationships must hold. In the left example, we infer that `*max` cannot alias with `*min`; while in the right example we infer that `kernel->positive_range` cannot alias with `kernel->values[i]`. Based on this additional information, the compiler register-allocates `*min` and `*max` for the full duration of the following loop (left example); similarly, the compiler unrolls the inner loop and vectorizes it in the right example. The resulting speedups in the two loops are 1.5x and 1.66x respectively, over Clang/LLVM's `-O3` compiled result (without our alias analysis) on a modern x86 machine.

## 2 Understanding C's OOE Specification

Before we can describe the problem, we need to formally specify the relevant C semantics. To do so, we reproduce relevant sentences from the C17 standard accompanied with associated small-step (reduction) operational semantic rules drawn from Norrish's work on abstract dynamic semantics for C [16]. Later efforts [6, 10] have further enriched Norrish's semantics with effective types and a full treatment of C features — our intent is not to discuss C semantics in their full glory; we just need to introduce the relevant portions required for our subsequent discussion on OOE-based optimization.

Our discussion on semantic rules is quite incomplete, e.g., we completely omit rules related to static type evaluation and statement evaluation, and focus exclusively on expression evaluation; even here, we omit many details (e.g., what is an rvalue context, what happens during type-casting, etc.). We refer the reader to Norrish's work [14, 16] for a fuller version of these operational semantic rules. We use almost the same notation as Norrish to facilitate easy cross-referencing. Our semantics are identical to that of Norrish's except for minor changes in the evaluation of the assignment operator and the post-increment/decrement operators, which stem from differences in the C11/C17 standard from the C89/C99 standard (for which Norrish's semantics were written).

### 2.1 Basics

*Some terminology used in the paper*: An expression may access a memory location as a read (also called reference) or a write (also called side effect). Two accesses *conflict* if they access the same location and at least one of them is a write. The evaluations of two subexpressions are *concurrent* if they can

proceed simultaneously without an intervening sequence point. We also call this pair of evaluations *unsequenced*. For example, in the expression `i = ++i`, the evaluation of the left operand (`i`) is concurrent with the evaluation of the right operand (`++i`) of the assignment operator. The situation of two concurrent and conflicting accesses is called an *unsequenced race*.

A *normal value* (also called *rvalue*) is a sequence of bytes associated with a type. Values resident in memory are called *objects*. An *lvalue* is a reference to an object (specified through a memory address and a type), and is allowed to "decay" into that object's rvalue through a memory reference.

An *expression* is a sequence of operators and operands that specifies the computation of a value (lvalue or rvalue), or that generates side effects (writes to memory), or both. The value computations of the operands of an operator are completed before the value computation of the result; however the same is not true for side effects. The evaluation of an expression uses a relation $\rightarrow_e$ (with reflexive transitive closure $\rightarrow_e^*$) that specifies the gradual transformation of expression syntax into an rvalue and side effects.

A *full expression* is a maximal expression that is separated from another expression through an *explicit sequence point*, which is typically something that ends with a semicolon or it could be a controlling statement of if/switch/while/do keywords. A full expression may internally involve multiple *implicit sequence points* based on the following rules: (1) There is a sequence point after evaluation of all function arguments and of the function designator, and before the actual function call. (2) There is a sequence point after evaluation of the first (left) operand and before evaluation of the second (right) operand of the following binary operators: && (logical AND), || (logical OR), and , (comma). (3) There is a sequence point after evaluation of the first (left) operand and before evaluation of the second or third operand (whichever is evaluated) of the ternary conditional operator '?:'.

A sequence point (explicit or implicit) defines any point in a program's execution at which it is guaranteed that all side effects of previous evaluations will have been performed, and no side effects from subsequent evaluations have yet been performed. We are interested in identifying the OOE of subexpressions of a single full expression, and thus for our purposes, all sequence points are implicit.

The reduction semantics for expression evaluation track the computed value and associated side effects. There may be multiple reductions for a given expression $e$ on an initial state $\sigma_0$ (represented as $\langle e, \sigma_0 \rangle$), and the abstract machine may nondeterministically pick any of them. To capture undefined behaviour, a special undefined value $\mathcal{U}$ is employed — C semantics state that if there *exists* an evaluation for $\langle e, \sigma_0 \rangle$ that leads to value $\mathcal{U}$, then $e$'s evaluation on state $\sigma_0$ is undefined. This also means that the abstract machine would never get "stuck": it would either transition to a well-defined value (and a final state), or to $\mathcal{U}$.

**Table 1.** Quick reference to symbols, operators and other miscellaneous notation used in section 2.

| | |
|---|---|
| **Syntax related and notational** | |
| $e$ | expression syntax |
| $\rightarrow_e$ | expression evaluation relation |
| $\rightarrow_e^*$ | reflexive transitive closure of $\rightarrow_e$ |
| $\sigma$ | machine state |
| $\sigma_0$ | initial state |
| $\mathcal{U}$ | undefined value |
| $\mathcal{E}$ | evaluation context (section 2.2) |
| $\odot$ | Binary operator (section 2.2) |
| $\square$ | Unary operator (section 2.2) |
| $\eta = \clubsuit(n, m)$ | side effect $\eta$ that writes value $m$ to address $n$ |
| $\hat{f}$ | identifies sequence point associated with function call |
| $\overset{\beta}{=}$, $\overset{\beta}{\odot=}$ | Simple and compound assignment operators maintain a bag of references $\beta$ made by either operand (section 2.8) |
| **Constructors** | |
| $\Lambda$ | (address $n$, type $t$) $\longrightarrow$ lvalue |
| $\mu$ | (constant value $n$, type $t$) $\longrightarrow$ bit pattern (section 2.3) |
| $R$ | constructor that ensures its value can only decay to an rvalue (section 2.4) |
| **State read functions** | |
| $\sigma^\tau$ | returns type of a variable in state $\sigma$ |
| $\sigma^\alpha$ | returns address of a variable in state $\sigma$ |
| $\sigma^v$ | (value $n$, type $t$) $\longrightarrow$ object of type $t$ at address $n$ in $\sigma$ |
| $\sigma^o$ | returns a function that calculates effect of (unary or binary) operator $o$ in $\sigma$ on its arguments; this is a state-based function because $o$ may operate on types defined in $\sigma$. |
| **State modification functions** | |
| `mark_ref` | records a memory reference |
| `remove_refs` | removes a recorded memory reference |
| `add_se` | records a side effect to memory; it is pending at this stage |
| `apply_se` | applies the pending side effect to memory (so it no longer remains pending) |
| `clear_se` | Clears the side effect and reference records |

For convenience, table 1 provides a quick reference to the symbols, operators, and miscellaneous notation introduced and used in this section.

### 2.2 Evaluation Context

Norrish employs an *evaluation context*, i.e., a piece of syntax with a "hole" in it, to succinctly specify the evaluation of an expression. The following rule uses the $\mathcal{E}$ context to capture ways in which evaluation of subterms may proceed

$$\frac{\langle e_0, \sigma_0 \rangle \rightarrow_e \langle e, \sigma \rangle}{\langle \mathcal{E}[e_0], \sigma_0 \rangle \rightarrow_e \langle \mathcal{E}[e], \sigma \rangle}$$

This rule states: "If expression $e_0$ can reduce to $e$, altering state $\sigma_0$ to $\sigma$, then the expression formed by inserting $e_0$ into the hole of the context $\mathcal{E}$ can start in $\sigma_0$, and reduce to the corresponding expression with $e$ in place of $e_0$, finishing in state $\sigma$." The allowed forms for $\mathcal{E}$ are

$$\mathcal{E}[\ \_\ ] ::= \_ \odot e \mid e \odot \_ \mid \square\_ \mid \ \_\&\&e \ \mid \ \_||e \ \mid \_, e \mid$$
$$\_.f \mid (t)\_ \mid \ \_?\, e_1 : e_2 \mid \_(e_1, ..., e_n) \mid e_0(e_1, .., \_, ..., e_n)$$

Here, $\odot$ stands for a standard binary operator:

$$\odot \in \{+, *, -, /, \%, \hat{}, |, \&, <<, >>, <, >, <=, >=, ==, !=\}$$

$\square$ stands for any of the unary operators $\in \{-, !, \sim, \&, *\}$. Notice that $\mathcal{E}$ encodes the unsequenced behaviour of many operators by allowing either operand to be evaluated first (e.g., for $\odot$). On the other hand, the &&, ||, comma, and ternary (?:) operators need to first evaluate their first operand. Also, some operators (e.g., assignment and pre/post-increment/decrement operators) are not captured by $\mathcal{E}$.

### 2.3 Base Cases

The base cases for expression evaluation are constants and variables. The following rule decays a constant identifier to its rvalue:

$$\frac{}{\langle (n,t), \sigma\rangle \rightarrow_e \langle \underline{(\mu(n,t),t)}, \sigma\rangle}$$

The function $\mu$ takes a value and a type and returns an appropriate bit-pattern for that type. The underlining indicates an rvalue that cannot be transformed any further.

Variables translate into lvalues as follows:

$$\frac{}{\langle \texttt{id}, \sigma\rangle \rightarrow_e \langle \Lambda(\sigma^{\alpha}(\texttt{id}), \sigma^{\tau}(\texttt{id})), \sigma\rangle}$$

For a state $\sigma$, $\sigma^{\tau}$ is a function that returns the type of the variable, and $\sigma^{\alpha}$ returns the address of the variable. $\Lambda$ is an lvalue constructor that takes the object's address and type as arguments. When used in an rvalue context, a non-array lvalue decays into its corresponding rvalue.

$$\frac{\Lambda(n,t) \text{ used in rvalue context ; } t \text{ not an array type}}{\langle \Lambda(n,t), \sigma\rangle \rightarrow_e \langle \underline{(\sigma^{v}(n,t),t)}, \texttt{mark\_ref}(\sigma, n, t)\rangle} \quad \text{(lvalue-to-rvalue)}$$

For any state $\sigma$, $\sigma^{v}(n,t)$ returns the object at address $n$ with type $t$. The $\texttt{mark\_ref}(\sigma, n, t)$ function takes a state $\sigma$, an address $n$, and a type $t$ and returns a new state which is identical to the first argument except that a memory reference to $(n,t)$ is additionally recorded in the new state. The record of memory references is maintained as a bag as part of the state. mark_ref is not a total function and can return the undefined value $\mathcal{U}$ if the reference was illegal (e.g., unallocated object); if so, the non-array lvalue expression evaluates to $\mathcal{U}$. Further, $\mathcal{U}$ propagates through the $\mathcal{E}$ context. $\mathcal{U}$ also propagates through other operators not included in $\mathcal{E}$ (e.g., assignment) similarly for all their operands (not shown).

$$\frac{}{\langle \mathcal{E}[\mathcal{U}], \sigma\rangle \rightarrow_e \langle \mathcal{U}, \sigma\rangle}$$

The lvalue of an array type can decay to a pointer rvalue without updating the bag of references (rule omitted).

### 2.4 Operators

The following rule obtains the lvalue for a **pointer dereference** ($*$), and the address-of (&) operator involves its inverse operation.

$$\frac{t \neq \texttt{Void}}{\langle *\underline{(\mu(n, t*), t*)}, \sigma\rangle \rightarrow_e \langle \Lambda(n,t), \sigma\rangle}$$

$$\frac{}{\langle \&(\Lambda(n,t)), \sigma\rangle \rightarrow_e \langle \underline{(\mu(n,t*), t*)}, \sigma\rangle} \quad \text{(address-of-lvalue)}$$

The **array dereference** operation $e_1[e_2]$ is treated identically to $*(e_1 + e_2)$.

For **structs**, a field dereference on a struct lvalue returns a new lvalue that has a different address (at field offset from the original address) and size (shrunk to field's size in bytes). The arrow operator `s->fld` is treated like `(*s).fld`. Functions are treated as variables of a function reference type where constants are the functions defined in the program (we omit the rules).

Value computation for **unary and binary operators** is performed after the value computations of the individual operand(s) is completed.

$$\frac{\odot \in \{+, *, -, /, \%, \hat{}, |, \&, <<, >>, <, >, <=, >=, ==, !=\}}{\langle \underline{(m_1,t_1)} \odot \underline{(m_2,t_2)}, \sigma\rangle \rightarrow_e \langle \underline{\sigma^{o}(\odot(m_1,t_1,(m_2,t_2)))}, \sigma\rangle}$$

$$\frac{\square \in \{-, !, \sim\}}{\langle \square\underline{(m,t)}, \sigma\rangle \rightarrow_e \langle \underline{\sigma^{o}(\square(m,t))}, \sigma\rangle}$$

Here, $\sigma^{o}$ is a function that calculates the effect of the operator ($\odot$ or $\square$) returning both value and type of the result.

Changes to memory happen solely through **side effects** captured in the state $\sigma$. For example, the assignment operator (=) evaluates to an rvalue (that of its right operand) and involves a side effect (update the object referred to by the lvalue of the left operand). However, side effects need not be applied immediately upon their creation, and can be kept pending until the next sequence point. Further, these pending side effects can be applied in any order. These semantics allow parallel or interleaved evaluation of subexpressions within an expression, and it is this low-level treatment of computation that gives C its special flavour. In Norrish's rules, this is modeled by keeping a bag of pending side effects as a part of the state. As the side effects are generated, they are put in the bag (shown while discussing side effect generating operators like assignment), and the bag is emptied in a non-deterministic order at non-deterministic times, subject to the constraints imposed by sequence points.

$$\frac{\eta \text{ is pending in } \sigma}{\langle e, \sigma\rangle \rightarrow_e \langle e, \texttt{apply\_se}(\sigma, \eta)\rangle}$$

Here, $\eta$ is a side effect, and apply_se applies the side effect to the memory state of the first argument. This application of a pending side effect can trigger at any time.

We now discuss an operator that involves a sequence point: the **comma** (,) operator. The evaluation of the second operand can start only after the first operand has been fully

evaluated, and all pending side effects have been applied.

$$\frac{\text{no pending side effects in } \sigma}{\langle \underline{(m,t)}, e \ , \ \sigma\rangle \rightarrow_e \langle R(e), \texttt{clear\_se}(\sigma)\rangle}$$

Here, `clear_se` clears the side effect and reference records (maintained as bags) of the state; the rule encodes that the evaluation can proceed across the comma operator (from the first operand to the second operand) only if there are no pending side effects in $\sigma$. $R$ is a constructor that ensures that its argument can decay only to an rvalue (and not to an lvalue), as captured by the only following rule that can remove $R$:

$$\frac{}{\langle R(\underline{(m,t)}), \sigma\rangle \rightarrow_e \langle \underline{(m,t)}, \sigma\rangle}$$

### 2.5 UB Associated With Unsequenced Evaluations

We now introduce the UB semantics associated with unsequenced evaluations (i.e., evaluations that do not have a sequence point between them). Quoting the C17 standard section 6.5 para 2: "If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behaviour is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behaviour is undefined if such an unsequenced side effect occurs in any of the orderings".

This C17 property renders the following two full expressions on the left (1 and 2) undefined while allowing the two full expressions in the middle (3 and 4). The two expressions on the right (5 and 6) may or may not be defined depending on whether `*p` and `i` are pointing to the same location or not.

```
1. i = ++i + 1;  |  3. i = i + 1;  |  5. *p = ++i + 1;
2. a[i++] = i;   |  4. a[i] = i;   |  6. a[i++] = *p;
```

In more detail, example 1 has no sequence points between the two updates to `i` and example 2 has no sequence points between the update to `i` and its use. In Example 4, there is no side effect (update) on `i`. Example 3 involves both an update and a use of `i`, however the two are sequenced because C17 states that the references made during the value computation of the assignment operation's operands are sequenced before the side-effect produced by that operation (section 6.5.16 para 3 of C17 standard). This property is also true for other side-effecting operators, namely compound-assignments, and pre/post-increment/decrement operators. We discuss this property in more detail while discussing the semantics of these side-effecting operators. Finally, evaluations of examples 5 and 6 will have well-defined behaviour only if `*p` and `i` are representing non-overlapping memory locations, and will have undefined behaviour if `*p` and `i` are representing the same memory locations.

We refer to the undefined behaviour caused by evaluations of the form specified by the quote in C17 standard's section 6.5.2 (reproduced above) as *unsequenced-UB*. Modeling this in Norrish's semantics involves stating rules that if for a state, the bag of references and the bag of side effects contain a conflicting address, then the `mark_ref` and the `apply_se` routines evaluate to $\mathcal{U}$, which in turn result in the expression evaluation to evaluate to $\mathcal{U}$.

Also, note that the second sentence in the quote reproduced above is significant; if there *exists* an evaluation path $\langle e, \sigma_0\rangle \rightarrow_e^* \langle \mathcal{U}, \sigma\rangle$, then $e$'s behaviour is undefined on $\sigma_0$.

$$\frac{\langle e, \sigma_0\rangle \rightarrow_e^* \langle \mathcal{U}, \sigma\rangle\rangle}{e\text{'s behaviour is undefined on } \sigma_0} \tag{1}$$

To see this with an example, consider the full expression "`(i--,j) + i;`". Here, there are at least two possible evaluations: one where the right use of `i` is evaluated *before* the comma operator is reached in the left subexpression; and another where it is evaluated after the comma is reached in the left subexpression. The UB occurs only in the second evaluation, but because there exists an evaluation to $\mathcal{U}$, the expression's behaviour is undefined.

### 2.6 Function Calls

A C function call also involves a sequence point after the evaluation of the function arguments and the function designator, and before the actual function call; this is captured through a special $\hat{f}$ identifier that represents that this sequence point is reached.

$$\frac{\begin{array}{c}\text{no pending side effects in } \sigma,\\ \text{all of } f \text{ and the } e_i\text{s are rvalues}\end{array}}{\langle f(e_1, ..., e_n), \sigma\rangle \rightarrow_e \langle \hat{f}(e_1, ..., e_n), \texttt{clear\_se}(\sigma)\rangle}$$

The special $\hat{f}$ identifier is then allowed to decay to the actual call. A C function call is special because it can read or write to memory without generating additional side effects and references: it simply updates the memory component of $\sigma$ without adding or subtracting from the bags of references or side effects. This means that the side effects and references performed within the callee function's body are not considered to race with the side effects and references of the caller's expression containing that function call. Consider the following C program:

```
int global = 0;
int foo() { return ++global; }
int main() { return foo() + (global=10); }
```

The underlined expression is *not* undefined because the invocation of `foo()` does not generate any references or side effects in the state (even though it updates `global`). However, the order of evaluation is still unspecified, and hence the (well-defined) result could be either 21 or 11 depending on the chosen order. Norrish proved that an expression that does not contain any function calls will always yield a deterministic result [15], and we state a similar theorem later in this section (theorem 2.1).

### 2.7 Short-circuiting Operators

The **logical binary operators** && and || involve "short-circuiting", in that their second operands are not always evaluated, e.g., if the first operand of || evaluates to `true`, the second operand is ignored:

$$\frac{m = \mu(1, t)}{\langle \underline{(m, t)} \;||\; e, \sigma\rangle \rightarrow_e \langle \underline{(\mu(1, \texttt{int}), \texttt{int})}, \sigma\rangle}$$

We omit a similar rule for &&. If these logical operators do not short-circuit (i.e., the value of the first operand forces the evaluation of the second operand), then computation proceeds similarly to how it proceeds for the comma operator (with a sequence point between the first and the second operand) – we omit the rules.

Next we discuss the **ternary conditional operator** $e_1 ? e_2 : e_3$ operator which involves a sequence point after the evaluation of $e_1$ and before the evaluation of *one* of $e_2$ or $e_3$, while ignoring the other.

$$\frac{\text{no pending side effects in } \sigma \qquad t \text{ is scalar} \qquad m \neq \mu(0, t)}{\langle \underline{(m, t)} ? \, e_1 : e_2, \sigma\rangle \rightarrow_e \langle e_1, \texttt{clear\_se}(\sigma)\rangle}$$

$$\frac{\text{no pending side effects in } \sigma \qquad t \text{ is scalar} \qquad m = \mu(0, t)}{\langle \underline{(m, t)} ? \, e_1 : e_2, \sigma\rangle \rightarrow_e \langle e_2, \texttt{clear\_se}(\sigma)\rangle}$$

### 2.8 Assignment and Pre/Post-Increment/Decrement

The **assignment operator** has two forms: simple ($x = y$) or compound ($x \odot= y$). In either form, it is possible for the evaluation of either operand to make references to the scalar object that is updated by the assignment operation (e.g., i=i+1), and we would like to allow this as well-defined behaviour. To capture this, Norrish maintains a bag of references $\beta$ made during the evaluation of either operand of the assignment operator (represented as $\overset{\beta}{=}$). This need to populate and maintain $\beta$ is the reason why the assignment operator was omitted from the general evaluation context $\mathcal{E}$.

$$\frac{t \text{ not an array type}}{\langle \Lambda(n, t) \overset{\beta}{=} \underline{(m, t)}, \sigma_0\rangle \rightarrow_e \langle \underline{(m, t)}, \sigma\rangle} \tag{2}$$

where $\sigma = \texttt{remove\_refs}(\sigma_1, \beta, n, t)$ and $\sigma_1 = \texttt{add\_se}(\sigma_0, \clubsuit(n, m))$. $\clubsuit(n, m)$ denotes a side effect that writes the value $m$ to address $n$. $\texttt{add\_se}(\sigma, \eta)$ adds the side effect $\eta$ to those pending in $\sigma$. $\texttt{remove\_refs}(\sigma, \beta, n, t)$ returns a state identical to $\sigma$ except that $\beta$'s recorded references to locations overlapping with $(n, t)$ are removed from the recorded references of $\sigma$. This removal of references to $(n, t)$ ensures that these references cannot be construed to conflict with the new side-effect generated by the assignment operator (e.g., this removal allows i=i+1).

The following rule describes the evaluation of the compound assignment operator $\odot=$ in terms of a simple assignment, only when the LHS has been reduced to its final lvalue:

$$\frac{}{\langle \Lambda(n, t_1) \;\overset{\beta}{\odot=}\; e_2, \sigma_0\rangle \rightarrow_e \langle \Lambda(n, t_1) \overset{\beta}{=} \Lambda(n, t_1) \odot e_2, \sigma_0\rangle} \tag{3}$$

The bag of references $\beta$ for compound assignments is updated during evaluation of the left and right operands, exactly as for simple assignments.

The evaluation of unary **pre/post-increment/decrement** operators also involves a side-effect, which also cannot be construed to conflict with the references produced during the value computation of the operand. Thus, similar to assignment, the set of references $\beta$ made by the operand is maintained; and during the reduction of the unary operator itself, the references in $\beta$ that conflict with the side-effect induced by the operator are removed. For example, the evaluation of the pre-increment operation (`++x`) proceeds identically to the evaluation of the compound assignment `x += 1`.

### 2.9 Deterministic Evaluation of C Expressions

Our alias-analysis algorithm is based on asserting the absence of UB during an expression evaluation. Before we describe our algorithm, we state a useful theorem.

**Theorem 2.1.** *For all expressions e, if e does not contain a function call, and if the evaluation of e is well-defined on an initial state $\sigma_0$ (i.e., $\langle e, \sigma_0\rangle$ does not evaluate to $\mathcal{U}$), then $\langle e, \sigma_0\rangle$ evaluates to a unique deterministic value and a unique deterministic final state for all possible (non-deterministic) evaluations.*

*Proof sketch.* The proof proceeds by induction on the structure of the expression. For deterministic operators, deterministic evaluations of operands would always yield deterministic results. For non-deterministic operators, the side effects of one operand are not allowed to conflict with the references or side effects of the other operand, which ensures that the two operands evaluate in isolation, i.e., one operand's evaluation cannot interfere with the other operand's evaluation. Thus the evaluation result of the full expression would be deterministic. Norrish provides a detailed argument for this theorem in his paper [15]. □

In section 3, we generalize this theorem to expressions that contains only *pure* function calls — a pure function does not read or write the global memory state (opposite of *impure*).

## 3 Algorithm

We now describe a static analysis algorithm that identifies must-not-alias relationships based on the semantics of C expression evaluation. In particular, we exploit the unsequenced-UB clause in the C specification (C17 standard's section 6.5.2 quote) to identify conflicting memory accesses that cannot alias for the expression evaluation to be well-defined. We use the following representation for an expression:

$$\underline{\texttt{id}} : \quad \text{“op”}(\underline{\texttt{id}}_1, \underline{\texttt{id}}_2, ..., \underline{\texttt{id}}_n)$$

where `id` is a unique identifier for every (sub)expression and "op" is a constructor that either represents an operator

or wraps a variable/constant (base case). For example, the representation of the expression `i = ++i + 10` is shown below (first read the left column from top to bottom, and then continue to the right column).

| | | | | |
|---|---|---|---|---|
| 0 | : | "var"(i) | 3 | : "+"(1, 2) |
| 1 | : | "++pre"(0) | 4 | : "var"(i) |
| 2 | : | "const"(10) | 5 | : "="(4, 3) |

The underlined values represent identifiers and the operators are indicated in quotes. "var" and "const" are constructors for variables and constants resp., and the "++pre"[2], "+", and "=" represent expression operators, operating on operands indicated through their identifiers.

Given an expression and its identifier id, we use judgements of the form $\texttt{id} : \omega, \theta, \gamma, \pi$ which is read as: *If the expression* id *is evaluated*,

- It generates at least $\omega$ memory references.
- It generates at least $\theta$ side effects.
- $\gamma \subseteq \theta$ are the side effects that are not followed by a sequence point in at least one evaluation of id.
- At least $\pi$ must-not-alias relationships can be inferred from id's evaluation.

$\omega$, $\theta$, and $\gamma$ are sets of lvalue sub-expressions of id, and $\pi$ is a set of unordered pairs of such lvalue sub-expressions. These lvalue subexpressions (named using their identifiers) represent memory objects that are either read ($\omega$) or written ($\theta$, $\gamma$); for every subexpression pair $(\texttt{id}_i, \texttt{id}_j) \in \pi$, it must be true that the evaluations of $\texttt{id}_i$ and $\texttt{id}_j$ cannot alias for the full expression evaluation to be well-defined on *any* initial state. Also note the antecedent of this judgement: these properties are true only if the expression gets evaluated.

The rules to compute sets $\omega$, $\theta$, $\gamma$, and $\pi$ are available in fig. 1. In general, any expression that evaluates to an lvalue and subsequently decays to an rvalue is added to $\omega$. If the lvalue is written-to (through one of pre/post-increment/decrement or simple/compound assignment operations), then it is added to both $\theta$ and $\gamma$; note that it is possible for the same subexpression to live in both $\omega$ and $\theta/\gamma$. Usually, these values $\omega$, $\theta$, and $\gamma$ of an expression are inductively constructed from the resp. values of the sub-expressions (through the set-union operator); however, if the top-level operation involves an implicit sequence point (e.g., due to a function call or a logical/ternary operator), $\gamma$ is cleared.

The set $\pi$ is populated at unsequenced operators (e.g., eq. (binop-unseq), eq. (fun-call)) by taking a cartesian product of the references/side-effects made by each sub-expression — these rules capture the property that the evaluation can proceed in either order for such operators and that there should be no unsequenced race in such evaluations. Notice that for operators that must decay their operand $e$ to its rvalue (eq. (unary-op), eq. (binop-unseq), eq. (deref), eq. (comma),

[2]Pre-increment (decrement) and post-increment (decrement) operators are represented as `++pre` (`--pre`) and `post++` (`post--`) resp.

$$\frac{}{(n,t) : \phi,\phi,\phi,\phi} \qquad \frac{}{\texttt{var\_id} : \phi,\phi,\phi,\phi} \qquad \text{(const / var)}$$

$$\frac{e_1 : \omega_1,\theta_1,\gamma_1,\pi_1 \qquad e_2 : \omega_2,\theta_2,\gamma_2,\pi_2 \qquad \odot \in \mathcal{N}}{\begin{aligned} e_1 \odot e_2 :\ & \omega_1 \cup \omega_2 \cup \nabla(\{e_1,e_2\}),\ \ \theta_1 \cup \theta_2,\ \ \gamma_1 \cup \gamma_2, \\ & \pi_1 \cup \pi_2 \cup \chi(\omega_1 \cup \nabla(\{e_1\}),\theta_2) \cup \chi(\theta_1,\omega_2 \cup \nabla(\{e_2\})) \\ & \cup \chi(\theta_1,\theta_2) \cup \chi(\gamma_1,\nabla(\{e_1\})) \cup \chi(\gamma_2,\nabla(\{e_2\})) \end{aligned}}$$
(binop-unseq)

$$\frac{e : \omega,\theta,\gamma,\pi \qquad \square \in \{-,!,\sim,(\texttt{rval}).\texttt{fld}\}}{\square e : \omega \cup \nabla(\{e\}),\theta,\gamma,\pi \cup \chi(\gamma,\nabla(\{e\}))}$$ (unary-op)

$$\frac{e : \omega,\theta,\gamma,\pi}{*e : \omega \cup \nabla(\{e\}),\theta,\gamma,\pi \cup \chi(\gamma,\nabla(\{e\}))}$$ (deref)

$$\frac{e : \omega,\theta,\gamma,\pi}{e.\texttt{fld} : \omega,\theta,\gamma,\pi}$$ (struct-field)

$$\frac{e : \omega,\theta,\gamma,\pi}{\&e : \omega,\theta,\gamma,\pi} \qquad \frac{e : \omega,\theta,\gamma,\pi}{\texttt{sizeof}\ e : \phi,\phi,\phi,\phi}$$ (address-of / sizeof)

$$\frac{e_1 : \omega_1,\theta_1,\gamma_1,\pi_1 \qquad e_2 : \omega_2,\theta_2,\gamma_2,\pi_2}{\begin{aligned} e_1,e_2 :\ & \omega_1 \cup \omega_2 \cup \nabla(\{e_1,e_2\}),\ \ \theta_1 \cup \theta_2,\ \ \gamma_2, \\ & \pi_1 \cup \pi_2 \cup \chi(\gamma_1,\nabla(\{e_1\})) \cup \chi(\gamma_2,\nabla(\{e_2\})) \end{aligned}}$$ (comma)

$$\frac{e_1 : \omega_1,\theta_1,\gamma_1,\pi_1 \qquad e_2 : \omega_2,\theta_2,\gamma_2,\pi_2 \qquad \odot \in \{\&\&,||\}}{e_1 \odot e_2 :\ \ \omega_1 \cup \nabla(\{e_1\}),\theta_1,\ \phi,\ \pi_1 \cup \chi(\gamma_1,\nabla(\{e_1\}))}$$
(binop-logical)

$$\frac{e_1 : \omega_1,\theta_1,\gamma_1,\pi_1 \qquad e_2 : \omega_2,\theta_2,\gamma_2,\pi_2 \qquad e_3 : \omega_3,\theta_3,\gamma_3,\pi_3}{e_1 ?\ e_2 : e3 : \omega_1 \cup \nabla(\{e_1\}),\theta_1,\phi,\pi_1 \cup \chi(\gamma_1,\nabla(\{e_1\}))}$$
(ternary)

$$\frac{e_i : \omega_i,\theta_i,\gamma_i,\pi_i}{\begin{aligned} e_0(e_1,e_2,\ldots,e_n) :\ & \bigcup_i(\omega_i \cup \nabla(\{e_i\})),\ \ \bigcup_i \theta_i,\ \ \phi, \\ & \bigcup_i(\pi_i \cup \chi(\gamma_i,\nabla(\{e_i\}))) \ \ \bigcup_{i,j:i\neq j} \chi(\theta_i,\theta_j) \ \ \bigcup \\ & \bigcup_{i,j:i\neq j}(\chi(\omega_i \cup \nabla(\{e_i\}),\theta_j) \cup \ \ \chi(\theta_i,\omega_j \cup \nabla(\{e_j\}))) \end{aligned}}$$
(fun-call)

$$\frac{e : \omega,\theta,\gamma,\phi,\pi \qquad \square \in \{\texttt{post}\ ++, \texttt{post}\ --, ++\texttt{pre}, --\texttt{pre}\}}{\square e : \omega \cup \{e\},\theta \cup \{e\},\gamma \cup \{e\},\pi \cup \chi(\{e\},\gamma)}$$
(pre/post-inc/dec)

$$\frac{e_1 : \omega_1,\theta_1,\gamma_1,\pi_1 \qquad e_2 : \omega_2,\theta_2,\gamma_2,\pi_2}{\begin{aligned} e_1 = e_2 :\ & \omega_1 \cup \omega_2 \cup \nabla(\{e_2\}),\ \ \theta_1 \cup \theta_2 \cup \{e_1\},\ \ \gamma_1 \cup \gamma_2 \cup \{e_1\}, \\ & \pi_1 \cup \pi_2 \cup \chi(\omega_1,\theta_2) \cup \chi(\theta_1,\omega_2 \cup \nabla(\{e_2\})) \cup \\ & \chi(\theta_1,\theta_2) \cup \chi(\{e_1\},\gamma_1 \cup \gamma_2) \cup \chi(\nabla(\{e_2\}),\gamma_2) \end{aligned}}$$
(assignment)

$$\frac{e_1 : \omega_1,\theta_1,\gamma_1,\pi_1 \qquad e_2 : \omega_2,\theta_2,\gamma_2,\pi_2}{\begin{aligned} e_1 \odot\!= e_2 :\ & \omega_1 \cup \omega_2 \cup \nabla(\{e_1,e_2\}),\ \ \theta_1 \cup \theta_2 \cup \{e_1\},\ \ \gamma_1 \cup \gamma_2 \cup \{e_1\}, \\ & \pi_1 \cup \pi_2 \cup \chi(\omega_1 \cup \{e_1\},\theta_2) \cup \chi(\theta_1,\omega_2 \cup \nabla(\{e_2\})) \cup \\ & \chi(\theta_1,\theta_2) \cup \chi(\{e_1\},\gamma_1) \cup \chi(\nabla(\{e_2\}),\gamma_2) \end{aligned}}$$
(compound-assignment)

**Figure 1. Static alias analysis algorithm**. The $\nabla(S)$ operator operates on a set of expression-ids $S$, and returns only those elements in $S$ that are non-array lvalues. $\phi$ represents the empty set. $e\dot{\texttt{f}}\texttt{ld}$ represents a struct field access, and the evaluated expression can either be an lvalue or an rvalue depending on $e$. $\chi(s_1,s_2)$ returns the cartesian product of sets $s_1$ and $s_2$. $\mathcal{N} = \{+,*,-,/,\%,\hat{},|,\&,<<,>>,<,>,<=,>=,==,!=\}$. $e_1[e_2]$ and `s->fld` are treated like $*(e_1 + e_2)$ and `(*s).fld` respectively.

**Table 2.** Representation of expression `*min = *max = a[0]`, and the computed $\omega$, $\theta$, $\gamma$, and $\pi$ sets.

| expression text | id | operation | $\omega$ | $\theta$ | $\gamma$ | $\pi$ |
|---|---|---|---|---|---|---|
| `a` | 0 | "var"(a) | {}, | {}, | {}, | {} |
| `0` | 1 | "const"(0) | {}, | {}, | {}, | {} |
| `a[0]` | 2 | "[]"(0, 1) | {}, | {}, | {}, | {} |
| `max` | 3 | "var"(max) | {}, | {}, | {}, | {} |
| `*max` | 4 | "*"(3) | {3}, | {}, | {}, | {} |
| `*max = a[0]` | 5 | "="(4, 2) | {2,3}, | {4}, | {4}, | {} |
| `min` | 6 | "var"(min) | {}, | {}, | {}, | {} |
| `*min` | 7 | "*"(6) | {6}, | {}, | {}, | {} |
| `*min = *max = a[0]` | 8 | "="(7, 5) | {2,3,6}, | {4,7}, | {4,7}, | {(4,7),(4,6)} |

eq. (binop-logical), eq. (ternary), eq. (fun-call), eq. (pre/post-inc/dec), eq. (assignment), and eq. (compound-assignment)), $\nabla(\{e\})$ is also considered a memory reference, and $\omega$ and $\pi$ are populated accordingly. Further, for these operators, we also infer must-not-alias relationships between $\nabla(\{e\})$ and the $\gamma$ set associated with the evaluation of $e$ — this captures the fact that side-effects generated during the evaluation of $e$ (but not followed by a sequence point) must not alias with lvalue $e$ (notice that $\nabla(\{e\})$ returns the empty set if $e$ does not evaluate to a non-array lvalue). The sizeof operator is unique because it may not actually compute the value of its operand, and we conservatively assume that it does not generate any memory references or side effects. Similarly, some operators (e.g., eq. (struct-field)) do not change any of our tracked values.

The rules of logical operators and ternary operators account for the fact that all subexpressions are not necessarily evaluated; e.g., depending on the value of the first operand, the second operand may or may not be evaluated. Thus, we conservatively only consider the memory references of only those subexpressions that will always get evaluated (e.g., the first operand of the logical/ternary operators).

Finally, the compound assignment involves unsequenced evaluation of both the first and the second operands into their rvalues, and thus $\pi$ is populated through a cartesian product of the respective $\omega$ and $\theta$ sets (similar to eq. (binop-unseq)). The rule for simple assignment has two important subtleties: (1) it is possible for the side-effect on the first operand $e_1$ to be always sequenced after the side-effects generated by the second operand (e.g., if the second operand involved a sequence point); thus, here the first operand $e_1$ is only paired with the $e_2$'s $\gamma_2$ (and not $\theta_2$) while populating $\pi$; (2) the first operand $e_1$ is not paired with $\omega_2$ while computing $\pi$ thus respecting the operation of `remove_refs` in eq. (2) (section 2.8).

To see the rules with an example, consider the expression `*min = *max = a[0]` (`a` is an array variable) with its representation in table 2. The $\omega$, $\theta$, $\gamma$, and $\pi$ sets for every subexpression of this expression are given in the last four columns above. Expression 4 (similarly 7) is the first subexpression where we record a memory access, populating $\omega$ with 3 (similarly 6). Expression 5 is an assignment and involves references to 2 and 3 by virtue of the references made by its first and second operands (including the second operand); it also involves a side-effect to the address evaluated through 4, but no must-not-alias relationships are inferred at this point[3]. Expression 8 involves references to {2, 3, 6} (union of references made by operands), and involves side effects {4, 7} (7 added due to the current assignment). Finally, this last assignment (8) allows us to infer that the two unsequenced side-effects (4 and 7) cannot alias; also the side effects of the right operand (4) and the references of the first operand (6) cannot alias for well-defined behaviour[4].

**Theorem 3.1.** *Consider an expression e such that $e : \omega, \theta, \gamma, \pi$.*

*1. if $e_1 \in \omega$ ($e_1 \in \theta$), then for all initial states $\sigma_0$, $e_1$ will evaluate to a non-array lvalue that has an associated memory reference (side-effect) in an evaluation of $\langle e, \sigma_0 \rangle$.*

*2. if $e_1 \in \gamma$, then $e_1 \in \theta$ and for all initial states $\sigma_0$, there exists an evaluation of $\langle e, \sigma_0 \rangle$ where the side-effect associated with $e_1$ has not been cleared at the time of adding the references/side-effects due to the top-level operator of e.*

*3. If $(e_1, e_2) \in \pi$, then for all initial states $\sigma_0$, the evaluation of $\langle e, \sigma_0 \rangle$ can result in a state where locations evaluated through lvalues $e_1$ and $e_2$ are accessed concurrently (i.e., without an intervening sequence point) and at least one of them is a write.*

*Proof sketch.* The proof for all three claims proceeds by induction on the structure of $e$.

For the first claim, we need to show that if $e_1 \in \omega$ (or $\theta$), then the evaluation must generate a reference through `mark_ref` (or side-effect through `add_se`) during the evaluation of $\langle e, \sigma_0 \rangle$. For short-circuiting operators, we only include the subexpressions that would always be evaluated (e.g., we ignore the second operand while computing the sets for the && operator).

For the second claim, we need to show that there exists an evaluation $\langle e, \sigma_0 \rangle$ where the side-effect on $e_1$ has not been cleared through `clear_se`. This is easy to show because we set $\gamma$ to empty set $\phi$ for all-but-one operators that involve a call to `clear_se`. The only exception is the comma operator,

[3]Recall that the references made during the value computation of either operand of the assignment operator are allowed to alias with the assignment operation's side effect.

[4]The references made by either operand of the assignment operator are allowed to alias with the side effect caused by the assignment operation, but not with the other operand's side effect.

where $\gamma$ is set to $\gamma_2$ because the evaluation of $e_2$ is guaranteed to follow (and not precede) the call to `clear_se` in this case.

For the third claim, we need to show that $(e_1, e_2)$ is added to $\pi$ only if concurrent read-write or write-write memory accesses to the two lvalues evaluated by $e_1$ and $e_2$ are possible in an evaluation of $\langle e, \sigma_0 \rangle$. Additions to $\pi$ occur either (a) through pairing an expression $e$'s memory access with the $\gamma$ set produced by $e$'s subexpressions (which is okay because $e$'s reference and $\gamma$'s side-effects can indeed be concurrent in an evaluation as stated in the second claim), or (b) for unsequenced operands of some operators ($\odot$, function call designator and arguments, assignments), by pairing the read-write and write-write memory accesses of the two operands. The latter is also okay because an unsequenced binary operator can evaluate its operands in either order thus guaranteeing the existence of an evaluation that can cause the two accesses to be concurrent. The only exceptions are the side-effecting operators (assignments, pre/post-inc/dec) where the references made during the value computations of either operand are sequenced before the side-effect induced by the side-effecting operator (section 2). To capture this, we never pair the references ($\omega$) produced by either operand with the side-effect induced by the side-effecting operator. □

**Theorem 3.2** (Soundness of rules in fig. 1)**.** *Consider an expression $e : \omega, \theta, \gamma, \pi$ that does not involve any function calls. Also consider any subexpression pair $(e_1, e_2) \in \pi$, and any arbitrary initial state $\sigma_0$. For $e$ to have well-defined behaviour on $\sigma_0$, it must be true that $e_1$ and $e_2$ do not alias in any evaluation of $\langle e, \sigma_0 \rangle$.*

*Proof sketch.* $e_1$, and $e_2$ are guaranteed to yield the same deterministic values and states in every evaluation of $\langle e, \sigma_0 \rangle$ (theorem 2.1). Further, there exists an evaluation (say $\alpha$) where accesses to addresses $e_1$ and $e_2$ proceed concurrently such that at least one is a side effect (theorem 3.1). Proof by contradiction: if $e_1$ and $e_2$ alias in some evaluation $\delta$, the $\alpha$ evaluation would also have aliasing values for $e_1$ and $e_2$ (because they are guaranteed to yield same values in every evaluation by theorem 2.1), and thus $\langle e, \sigma_0 \rangle$ would evaluate to $\mathcal{U}$ in $\alpha$, making $e$'s behaviour undefined on $\sigma_0$ (eq. (1) in section 2.5). □

Theorem 3.2 is significant because it means that for *all* evaluations of $e$ over initial state $\sigma_0$, and for all $\sigma_0$, $e_1$ and $e_2$ cannot alias. This allows us to generate a corresponding must-not-alias predicate (for every element of $\pi$) and expose it to the compiler for optimization.

If an expression $e$ involves function calls however, the above theorem and proof do not apply (because theorem 2.1 does not apply to such expressions) and a counter-example is given in table 3 (program shown on left, unsequenced OOE expression shown on right).

**Table 3.** Counter-example involving function calls

```
int a = 0,b = 2;
int *foo() {
  if (a == 1) return &a;
  else return &b;
}
int main() {
  return (a = 1) + *foo();
}
```

```
0 : "var"(a)
1 : "const"(1)
2 : "="(0, 1)
3 : "const"(foo)
4 : "call"(3)
5 : "*"(4)
6 : "+"(2, 5)
```

We first note that this program's behaviour is well-defined because for all possible evaluation orders, there is no situation where two conflicting memory accesses are unsequenced. However, when we run our algorithm on the underlined expression, we obtain a predicate that incorrectly states that the side effect to the lvalue evaluated through `p` must not alias with the reference during the evaluation of the expression `*foo()`, as we discuss next. Running our algorithm in fig. 1 on the top-level expression (id 6), the subexpression 2 would have $\theta = \{0\}$ (due to assignment); and so while computing $\pi$ for expression 6, our rules in fig. 1 would identify 0 and 5[5]. as must-not-alias pairs, which is incorrect — in some evaluations it is possible for 0 and 5 to alias and yet it would be well-defined because the aliasing accesses would be separated by a sequence point (call to `foo`). The problem arises because the evaluation $\alpha$ in which the two accesses (to 0 and 5) are performed concurrently may not coincide with the evaluation $\delta$ in which they evaluate to the same values, and thus the arguments made in theorem 3.2 are no longer valid.

For expressions with function calls, the difficulty arises because the callee function may read or write to the locations being accessed in the expression without generating any references or side effects. To support expressions with function calls, we add an additional overriding rule to our algorithm in fig. 1:

$$\frac{\forall_i e_i : \omega_i, \theta_i, \gamma_i, \pi_i \qquad \exists_i e_i \text{ contains impure fcall}}{\mathtt{OP}(e_0, \ldots, e_n) : \ \omega_{orig}, \ \ \theta_{orig}, \ \ \gamma_{orig}, \ \ \pi_{orig} \cap (\cup_i \pi_i)} \text{(impure-fun-call)}$$

Here `OP` represents any C operator, and $\omega_{orig}$, $\theta_{orig}$, $\gamma_{orig}$, and $\pi_{orig}$ represent the original values computed through the algorithm in fig. 1. This rule states that if any of the sub-expressions contain an impure function call (i.e., a function call that can read/write global memory), then no new elements are added to $\pi$ due to `OP`.

**Theorem 3.3** (Soundness in presence of function calls)**.** *Consider an expression $e : \omega, \theta, \gamma, \pi$ that may contain function calls. Also consider any subexpression pair $(e_1, e_2) \in \pi$ (as computed by rules in fig. 1 including the overriding rule in eq. (impure-fun-call)), and any arbitrary initial state $\sigma_0$. For $e$ to have well-defined behaviour on $\sigma_0$, it must be true that $e_1$ and $e_2$ do not alias in any evaluation of $\langle e, \sigma_0 \rangle$.*

[5] 5 is added due to the term $\nabla(\{e_2\})$ in eq. (assignment) in fig. 1.

*Proof sketch.* In eq. (impure-fun-call), predicates are added to $\pi$ only if no impure function calls are present in the current subexpression. The first part of the proof involves showing that these subexpressions (at which $\pi$ is populated) will always evaluate to a deterministic result for a given initial state. This can be shown through a straight-forward extension of theorem 2.1, where we need an extra argument at the inductive step showing that a call to a pure function with deterministic arguments can only yield a deterministic result, and that the pure callee function will not update memory state. The second part of the proof is identical to the proof for theorem 3.2. □

# 4 Evaluation

## 4.1 Implementation

We have implemented our algorithm inside Clang/LLVM, after the stage where the Abstract Syntax Tree (AST) has been formed for the program being compiled, and before the LLVM IR is generated. We compute the must-not-alias relationships ($\pi$) as explained in section 3 and encode these relationships through LLVM's intrinsic calls emitted into the LLVM IR code just before the code for evaluating the respective expression (which uses a deterministic OOE). These (debug) intrinsic calls encode the required must-not-alias predicates and we have ensured that they do not otherwise affect the compiler's optimizations in any way. For each expression with unsequenced side-effects, multiple intrinsic calls may be generated, one for each must-not-alias predicate (element in $\pi$). Then we wrote a custom analysis that parses these intrinsic calls and exposes these relationships to LLVM's alias-analysis subsystem. Because the LLVM optimizers query the alias-analysis subsystem from time to time, the results of our analysis become available to the optimizers while they make transformation decisions.

The implementation is slightly complicated by the fact that we need to distinguish between pure and impure callee functions during alias analysis. To handle this, our AST-based analysis *tags* each must-not-alias predicate with the function calls that appear in either expression of that predicate. Initially we (conservatively) assume that all these function calls are impure, and thus for the predicates that include an impure function call, we do not expose them to the alias-analysis subsystem. As LLVM's analysis passes proceed, LLVM's function summaries become available, and we distinguish between pure and impure functions by looking at the `readnone` attribute in these summaries. Predicates involving calls to only pure functions are then additionally exposed to the alias-analysis subsystem.

Exposing the must-not-alias predicates to the alias-analysis subsystem involves adding an additional alias-analysis (`aa`) algorithm, which we call `unseq-aa`. LLVM has multiple such algorithms (`basic-aa`, `type-based-aa`, etc.), each of which can be queried to obtain the strongest aliasing relationship that is known for the two lvalues that form the query arguments. The response of an algorithm could be one of may-alias, must-alias, partial-alias, or must-not-alias. LLVM queries each algorithm in series, stopping at the first algorithm that returns must-not-alias. For evaluation, we count the number of times `unseq-aa` returned must-not-alias when all other `aa` algorithms in LLVM return `may-alias`. It is worth noting that as optimizations proceed, the LLVM values (including the arguments of our intrinsic calls encoding must-not-alias predicates) may get modified or eliminated — to handle these possibilities, we wrap the arguments of our intrinsic calls inside "metadata nodes". This ensures that these references to LLVM values are not a part of the actual program computation and so it is acceptable for LLVM passes to eliminate them.

Apart from identifying the optimization opportunity through a more powerful alias analysis, we have also implemented a UB-sanitizer that instruments the program with runtime assertion checks to ensure that the program behaviour is not undefined for the provided inputs. LLVM developers usually follow a discipline wherein any optimization that leverages UB should also be accompanied by a corresponding sanitizer. In the same vein, our sanitizer converts the must-not-alias predicates ($\pi$) into runtime checks asserting that the two arguments of these predicates do not alias. Unlike optimizations, where we interpose on all stages of compilation; for our sanitizer, we limit ourselves to the unoptimized LLVM IR. Because some of the information about function's "purity" is available only in the later compilation stages, for our sanitizer, we conservatively generate predicates for only those must-not-alias relationships where none of the expressions contain a function call: in our experiments, these constitute > 98.5% of the total must-not-alias relationships generated at the AST level.

## 4.2 Experiments

The experiments were performed on a system with Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz processor. The L1d cache, L1i cache, L2 cache and L3 cache sizes were 32K, 32K, 256K and 2048K respectively. For comparison, we use recent versions of the GCC (v5.4.0), Clang/LLVM (v8.0.0), and ICC (v19.0.1.144) compilers with `-O3` enabled.

**Table 4.** Speedups on annotated Polybench functions.

| `bicg` | `gesummv` | `jacobi-1d` | `gemm` | `atax` | `trisolv` |
|---|---|---|---|---|---|
| 2.62x | 2.31x | 1.69x | 1.11x | 1.10x | 1.06x |

**4.2.1 Polybench Results.** We first discuss the results of applying OOElala on programs drawn from the Polybench benchmark suite [2], where we manually modified the benchmarks' source code to introduce expressions with unsequenced side-effects to allow the compiler to infer the necessary must-not-alias relationships. Here is an example

**Table 5.** Statistics obtained by OOElala on SPEC CPU2017 programs

| Name | kloc | # initial full exprs unseq. s.e. | # initial preds | # final preds | # unique final preds | additional must-not-alias responses | additional may-alias responses | additional must-alias responses | % increase in # alias queries |
|---|---|---|---|---|---|---|---|---|---|
| gcc | 1304 | 30125 | 86950 | 12427 | 5894 | 101861 | 309357 | 12389 | 0.12 |
| x264 | 96 | 1458 | 6999 | 11059 | 6537 | 6749 | 62094 | 43 | 0.28 |
| perlbench | 362 | 3768 | 7169 | 10616 | 5451 | 6352 | 612206 | -13 | 0.43 |
| xz | 33 | 505 | 778 | 524 | 383 | 2452 | 614 | 106 | 0.55 |
| imagick | 259 | 2585 | 3453 | 6627 | 1685 | 960 | 25556 | 362 | 0.01 |
| nab | 24 | 124 | 292 | 596 | 183 | 93 | -93 | 0 | 0 |
| mcf | 3 | 62 | 74 | 90 | 26 | 0 | 0 | 0 | 0 |
| lbm | 1 | 36 | 36 | 36 | 36 | 0 | 0 | 0 | 0 |

of a manual annotation performed by us for Polybench's `bicg` kernel:

```
#define CANT_ALIAS(a,b,c,d,e) ((a = a) & (b = b) \
                & (c = c) & (d = d) & (e = e))
void kernel_bicg(....) { ....
   for ( .... ) {
     ...; CANT_ALIAS(s[j], r[i], A[i][j], q[i], p[j]); ...
   } .......           }
```

The `CANT_ALIAS` preprocessor macro constructs a no-op expression with unsequenced side-effects, to encode the must-not-alias relationships among its arguments. The macro can be used at any point in the program, and for any (number of) scalar program values as arguments. This is in contrast with C's `restrict` keyword which suffers from an all-or-nothing problem, e.g., either a pointer may alias with everything, or (if restrict qualified) it must not alias with anything[6]. Further, modern compilers find it hard to exploit the relationships encoded by `restrict` in general, e.g., the current LLVM-8.0.0 implementation only exploits `restrict` when it is used with function arguments and ignores the `restrict` keyword when it is used in the function body [1]. We run each benchmark five times and take the median. With only a few lines of `CANT_ALIAS` annotations per Polybench function, we recorded speedups of up to 262.3%, when compiled with OOElala. On the other hand, these annotations made no difference to benchmark performance when compiled with other compilers, namely GCC, Clang/LLVM, and ICC. In table 4, we list some Polybench functions and the associated speedups we obtained on them due to our annotations with OOElala. These speedups are a direct result of improved optimization made possible due to the additional available must-not-alias relationships.

**4.2.2 SPEC CPU 2017 Results.** We next discuss the results of applying our algorithm to the (unmodified) SPEC CPU 2017 benchmarks. We compiled the eight C-only benchmarks available as part of the Intrate and FPrate suites with `O3` optimization flag.

**OOElala statistics**: We show the statistics obtained by OOElala in table 5. The first column is the benchmark name and the second column (# kloc) indicates the number of kilo lines of code in the benchmark. The third column (# initial full exprs unseq. s.e.) indicates the number of full expressions in the original AST that involve at least one unsequenced side effect and access (and thus generate at least one predicate). The fourth column indicates the number of must-not-alias predicates that were generated by our algorithm when applied on the original AST (including the ones that are tagged with impure callee function names). Recall that of these generated predicates, only those are exposed to the alias-analysis subsystem that contain either none or only pure function calls. The fifth column indicates the number of predicates remaining after all O3 optimization passes have been applied, the predicates which involve calls to impure functions have been removed and the predicates with both the operands as bitfields have been ignored (discussed in section 4.2.3). Notice that the number of remaining predicates can be higher than the number of initial predicates (even though we are removing some predicates) because transformations like unrolling and inlining make multiple copies of the same predicate. The sixth column counts the number of "unique" predicates after all O3 optimization passes, where multiple copies of the same predicate (due to unrolling for example) are counted as one. The seventh column indicates the number of additional must-not-alias responses returned by `aa-eval` during LLVM's O3 transformation pipeline – this is the number of must-not-alias responses returned by `unseq-aa` when all other LLVM alias analysis algorithms would return may-alias. As a consequence of these extra must-not-alias responses, the compiler often executes transformations like unrolling, inlining, etc., which could in turn increase the total number of `aa-eval` queries made, thus also increasing the number of may-alias and must-alias responses (eighth and ninth columns). Over 40k extra must-not-alias predicates are generated for these eight programs through our algorithm, that result in over 100k extra must-not-alias responses that would have otherwise been decided as may-alias responses. The last column indicates the percentage increase in the number of alias queries performed by OOElala as compared to LLVM.

Our algorithm produces at least one extra must-not-alias predicate in 720 of the 772 files compiled across these eight benchmarks and it provides more aliasing information (in terms of queries) to the compiler in 165 of these 720 files.

[6] A similar proposal arguing the redundancy of the `restrict` keyword in C has previously appeared as a blog post: http://blog.frama-c.com/index.php?post/2012/07/25/On-the-redundancy-of-C99-s-restrict.

```
//500.perlbench_r regexec.c
#define SSPOPINT (PL_savestack[--PL_savestack_ix].any_i32)
#define SSPOPIV (PL_savestack[--PL_savestack_ix].any_iv)
void S_regcppop(pTHX_ regexp *rex, U32 *maxopenparen_p) {
  *maxopenparen_p = SSPOPINT; ...      // line 364
  for ( ; i > 0; i -= REGCP_PAREN_ELEMS) { // line 377
  rex->offs[paren].start = SSPOPIV; ... } ...
} // SE on PL_savestack_ix is unseq WRT to SE on
// *maxopenparen_p, and read of maxopenparen_p. DSE can
// remove multiple stores to PL_savestack_ix. The SE is
// also unseq WRT read of paren and rex->offs[paren],
// and LICM can hoist loads and sink stores to it.
// Improvement: 4.71% (250k calls)
```

```
//502.gcc_r omega.c line 2099
for (e = pb->num_geqs - 1; e >= 0; e--) {
  int tmp = 1;  is_dead[e] = false;
  peqs[e] = zeqs[e] = neqs[e] = 0;
  for (i = pb->num_vars; i >= 1; i--) {
    if (pb->geqs[e].coef[i] > 0) peqs[e] |= tmp;
    else if (pb->geqs[e].coef[i] < 0) neqs[e] |= tmp;
    else zeqs[e] |= tmp; ...
} } // SE on peqs[e] is unseq WRT SE on zeqs[e], and
// SE on neqs[e]. LICM pass can
// register allocate peqs[e], neqs[e], zeqs[e] and
// move their loads/stores outside the inner loop
```

```
// 500.perlbench_r toke.c line 8585
do {
*(*d)++ = *(*s)++;
} while (isWORDCHAR_A(**s) && *d < e);
// SE on *d is unseq WRT SE on *s
// LICM pass can register allocate *d
// and *s, and sink stores to them outside the loop
// body. Improvement: 5.33% (20k calls)
```

```
// 502.gcc_r regmove.c line 1277
for(op_no=recog_data.n_operands;--op_no>=0;)
  matchp->with[op_no]=matchp->commutative[op_no]=-1;
// SE on matchp->commutative[op_no] is unseq WRT SE
// on matchp->with[op_no]. The SE's are also unseq
// WRT read of matchp. LoopVectorize pass can use
// this in its cost calculation and perform greater
// partial unrolling, leading to greater instruction
// level parallelism. Improvement: 2.46% (502k calls)
```

```
//557.xz_r delta_encoder.c line 27
for (size_t i = 0; i < size; ++i) {
   const uint8_t tmp = coder->history[
   (distance + coder->pos) & 0xFF];
    coder->history[coder->pos-- & 0xFF] = in[i];
    out[i] = in[i] - tmp;
} // SE on coder->pos is unseq WRT read of
// coder->history, and read of in[i]. LICM pass is
// able to register allocate code->pos, and
// sink the stores to it outside the loop
```

```
// 502.gcc_r: cfglayout.c line 683
FOR_ALL_BB (bb) {
  bb->aux = NULL;   bb->il.rtl->visited = 0;
  if (!stay_in_cfglayout_mode)
    bb->il.rtl->header = bb->il.rtl->footer = NULL;
} // SE on bb->il->header is unseq WRT SE on
// bb->il->footer. Both SE's are also unseq WRT
// read of bb->il. MemCpyOpt and MemDep passes
// replace two stores to header and footer
// by single memset. Improvement: 2.05% (14k calls)
```

```
//557.xz_r range_encoder.c line 105
do {
  rc->symbols[rc->count++]
    = RC_DIRECT_0 + ((value >> --bit_count) & 1);
} while (bit_count != 0);
// SE on rc->count is unseq WRT SE on bit_count, and
// read of rc->symbols. LICM pass can register-
// allocate rc->count and later LoopVectorize can
// perform loop-versioning
```

```
// 525.x264_r : io_tiff.c line 222
static uint32 getU32 (Tiff * t) {
  union { uint8 in[4]; uint32 out; } u;
  u.in[0] = *t->mp++;  u.in[1] = *t->mp++;
  u.in[2] = *t->mp++;  u.in[3] = *t->mp++;
  return u.out;
} // SE on t->mp is unseq WRT read of *t->mp.
// DSE pass and MemDep can remove intermediate
// stores to t->mp, keeping the final one
```

**Figure 2.** Some patterns discovered through OOElala in SPEC CPU 2017 programs.

**Case studies**: In fig. 2, we show some more code snippets from the SPEC CPU 2017 benchmarks demonstrating the coding patterns that allow OOElala to identify additional must-not-alias relationships. With each code snippet in fig. 2, we also show (a) the additional optimization passes that are enabled, (b) the number of times that code snippet is executed during a run of the SPEC programs on the provided reference inputs, and (c) the runtime improvement for each snippet on these inputs. These runtime improvement statistics were collected by instrumenting these code snippets with the `clock()` library function. For four of these snippets (x264's `io_tiff.c`, gcc's `omega.c`, xz's `delta_encoder.c`, and xz's `range_encoder.c`), we find that they are not executed even once for the provided reference inputs. For the other five, we notice improvements of 66% (imagick's `morphology.c`), 5.33% (perlbench's `toke.c`), 4.71% (perlbench's `regexec.c`), 2.46% (gcc's `regmove.c`) and 2.05% (gcc's `cfglayout.c`). We briefly outline the new transformations performed in these examples as a result of our analysis algorithm:

- In `perlbench_r regexec.c`, the side-effect (SE) on `PL_savestack_ix` is unsequenced with the read and SE on `*maxopenparen_p`. Thus the *dead-store elimination* (`DSE`) pass is able to remove duplicate stores to `PL_savestack_ix` based on the must-not-alias information thus generated. Similarly, there is another unsequenced expression in the same file where reads to `paren` and `rex->offs[paren]` are unsequenced with the SE on `PL_savestack` and `PL_savestack_ix`, and so this allows the *loop-invariant code motion* (`LICM`) pass to hoist loads and sink stores to `rex->offs[paren]`.
- In `gcc_r omega.c`, the three unsequenced SEs allow `LICM` to register-allocate `peqs[e]`, `neqs[e]` and `zeqs[e]`, and move their loads/stores outside the inner for-loop.
- In `perlbench_r toke.c`, the extra must-not-alias relationship allows `LICM` to register-allocate `*d` and `*s` and sink stores to them outside the loop.

**Table 6.** SPEC CPU 2017 performance results

| Name | Time (s) (clang) | Score (clang) | Time (s) (OOElala) | Score (OOElala) | Improvement | Std. dev. for clang (% of mean) | Std. dev. for OOElala (% of mean) |
|---|---|---|---|---|---|---|---|
| gcc | 367.22 | 3.856 | 367.05 | 3.858 | 0.052% | 0.06 | 0.1 |
| x264 | 379.05 | 4.619 | 376.03 | 4.656 | 0.794% | 0.03 | 0.05 |
| perlbench | 476.80 | 3.339 | 479.24 | 3.322 | -0.511% | 0.47 | 0.26 |
| xz | 477.49 | 2.261 | 478.14 | 2.259 | -0.088% | 0.27 | 0.13 |
| mcf | 569.75 | 2.836 | 570.34 | 2.833 | -0.106% | 0.02 | 0.03 |
| imagick | 615.56 | 4.040 | 612.79 | 4.058 | 0.443% | 0.08 | 0.5 |
| nab | 478.93 | 3.514 | 480.53 | 3.502 | -0.343% | 0.25 | 0.04 |
| lbm | 688.29 | 1.531 | 686.28 | 1.536 | 0.325% | 0.01 | 0.12 |
| **Overall** | | **3.089** | | **3.091** | **0.064%** | **0.22** | **0.22** |
| **Overall (w/o perlbench)** | | **3.055** | | **3.060** | **0.147%** | **0.14** | **0.20** |

- In `gcc_r regmove.c`, the two SEs are unsequenced; the *loop vectorization* (`LoopVec`) pass uses the extra aliasing information in its cost calculation and performs greater partial unrolling for greater Instruction-Level Parallelism.
- In `xz_r delta_encoder.c`, based on the extra information, `LICM` is able to register-allocate `code->pos` and sink the stores to it outside the loop.
- In `gcc_r cfglayout.c`, the *memcpy optimization* (`MemcpyOpt`) and the *memory dependence analysis* (`MemDep`) passes together replace the two stores to `header` and `footer` fields by a single `memset`.
- In `xz_r range_encoder.c`, `LICM` register-allocates `rc->count` and `LoopVec` performs loop versioning.
- In `x264_r io_tiff.c`, `DSE` and `MemDep` together remove intermediate stores to `t->mp` retaining the final one.

Through examining the compile-time statistics for some of the files where these patterns occur, we make some interesting observations: (1) `imagick`'s `morphology.c`: Number of loops vectorized increases by 3 (from 11). (2) `x264`'s `io_tiff.c`: Number of nodes combined during code generation through LLVM's SelectionDAG increases by 294 (from 1549). (3) `perlbench`'s `regexec.c`: Number of instructions (including loads) either simplified or eliminated through LLVM's GVN increases by 387 (from 6831), the number of inlined function calls increases by 6 (from 226), and the number of deleted functions increases by 1 (from 20). (4) `perlbench`'s `toke.c`: Number of loops unrolled and vectorized increases by 1 (from 3). (5) `xz`'s `delta_encoder.c`: Number of loops unrolled increases by 2 (from 2), and the number of registers assigned during register allocation increases by 66 (from 235). (6) `xz`'s `lzma_encoder.c`: Number of LICM hoistings increase by 9 (from 47). These statistics indicate that our must-not-alias predicates aid compiler transformations to help improve code performance.

**Overall performance improvement**: To understand the overall performance impact of this analysis, we also ran the SPEC benchmarks on the provided reference inputs with and without our alias-analysis algorithm. Each benchmark program was run for three iterations for each input and the base run metrics were collected whose median value was selected. The runtimes and corresponding SPEC scores and the corresponding standard deviation values are tabulated in table 6 for baseline Clang/LLVM and OOElala. The overall runtime performance improvement due to OOElala for the SPEC benchmark suite was recorded as 0.064%. The increase in compilation time due to OOElala is $< 2\%$.

It was surprising to see that `perlbench` became slower by about 0.5%. To investigate this issue further, we ran Intel's vTune Amplifier to profile `perlbench` and identified that the function `S_regmatch` is responsible for the slowdown as it reports a 16.66% increase in the number of cycles spent on instruction-cache (icache) misses. We find that this is because of excessive inlining performed by LLVM: our additional must-not-alias responses result in further optimizations in the `S_regcppop` function, as discussed in fig. 2, which lead to a shorter implementation. This causes the function to fit within LLVM's inlining threshold and thus all calls to it are completely inlined now. While such inlining should usually be an optimization, on this particular machine, it resulted in a larger number of icache misses. To fix this, one may perhaps need to revisit LLVM's inlining logic and thresholds, which is beyond the scope of this paper. If we ignore `perlbench`, the overall performance improvement is 0.147%.

#### 4.2.3 UB Sanitizer results.

Finally, we ran the implemented UBSan checks on the SPEC programs without optimizations on the provided reference inputs. This exercise led us to discover an implementation-level subtlety related to LLVM's widening of bitfield objects (objects which are smaller than a single byte) to their byte-level objects. This coarse-grained treatment of bitfields may result in the generation of unsound predicates during the lowering to LLVM IR because two different bitfields that belong to the same byte would now have identical addresses and our predicate would incorrectly encode a must-not-alias relationship between the identical byte addresses. To preserve soundness, we conservatively ignore all predicates both of whose expressions are bitfields. After this fix to our implementation, all eight programs ran successfully with the UBSan runtime checks enabled, indicating that the programmers' usage of OOE non-determinism is likely correct.

## 5 Related Work

There exist formal operational [7, 14, 16], denotational [17], and axiomatic [10] semantics of C's expression evaluation, and all such efforts carefully handle the quirks of the UB associated with unsequenced memory accesses. To our knowledge, ours is the first effort to exploit these UB semantics for optimization. We find that real-world code presents several optimization opportunities based on these C semantics. Ellison and Rosu [6] contributed an executable formal semantics of C11. Later work by Hathhorn et. al. [8] carefully developed "negative" semantics of the C11 language, wherein undefined evaluations are rejected explicitly by the verification tool. Both these efforts carefully model unsequenced-UB in their semantics. Our UBSan extension is also a limited form of "negative" checks, albeit weaker — while Hathhorn et. al.'s semantics can catch unsequenced-UB if it occurs on the current evaluation being performed, our UBSan extension catches unsequenced-UB only if it would have occurred in all possible evaluations. On the other hand, we are able to run our checks on over two million lines of code (SPEC CPU2017 programs), whereas Hathhorn et. al. tested their tool on much smaller GCC torture tests.

CompCert [11] uses a fixed left-to-right evaluation order for C expressions, but also provides a `-all` flag in its C interpreter that attempts all possible evaluation orders and returns the result for each order. CompCert can catch certain types of UB that are encountered but it does not support the identification of unsequenced-UB, and would silently ignore it. Blazy and Leroy [4] present mechanized big-step operational semantics for the Clight subset of the C language. Clight does not allow expressions to have any side effects, thus side-stepping OOE non-determinism. Batty et. al. [3] formalize the dynamic C++ concurrency semantics which include concurrency introduced due to unsequenced expression evaluations. Morisset et. al. [13] employ these semantics to further prove the correctness of program transformations. Both these works are primarily interested in either specifying concurrency semantics or verifying existing transformations against these semantics, but do not attempt to identify additional optimizations enabled by these semantics.

Wang et. al. [18] demonstrated a tool that can statically detect "optimization unstable" code, i.e., likely-buggy code patterns that can be optimized away through UB-exploiting compiler optimizations. While their work exposes the harmful side of UB non-determinism and provides a static checker to mitigate these harmful effects (similar to our UBSan tool, although UBSan is dynamic), we show that an aware programmer can, in fact, exploit OOE-based UB for optimization.

Design discussions for modern languages have made a case for minimizing non-determinism to aid programmability, debuggability, reproducibility, testing and verification. However there is much less discussion on the performance aspects of these non-deterministic choices. Clearly, non-determinism has both merits and demerits which need to be evaluated in equal measure. Merits of non-determinism are usually related to optimization potential and intuitiveness of usage by programmers. For example, our usage results on SPEC benchmarks indicate that programmers are using unspecified OOE in C correctly. On the other hand, unspecified OOE seems to have caused much programmer pain in C++ [5]. We argue that such questions need to be answered separately for each programming language through detailed quantitative studies on usage and optimization opportunity.

Dataflow-based alias analysis techniques have been well-studied in research literature [9]; in contrast, we exploit OOE non-determinism to infer must-not-alias relationships, and the aliasing relationships we derive are often not identifiable through (even the most powerful) dataflow analyses.

Mock [12] studied the obtainable speedups through programmer-specified aliasing annotations using C99's `restrict` keyword for SPEC benchmarks. These `restrict` annotations were limited to the function arguments in this work. Mock reported up to 7.5% speedups through automated (but not necessarily sound) `restrict` annotations in the source code of the SPEC programs. In our experiments on the SPEC benchmarks, we have not introduced any extra `CANT_ALIAS` annotations, and our improvements are entirely due to the information obtained from existing expressions in these programs. It is likely possible to obtain better speedups by adding `CANT_ALIAS` annotations to the SPEC source, either manually or algorithmically. In contrast to `restrict`, the `CANT_ALIAS` macro enables finer-grained specification of must-not-alias relationships and can potentially be used at arbitrary program points. This makes it both easier to use for the programmer, and more potent from an optimization perspective. Also, the compiler-side support required to exploit these `CANT_ALIAS` annotations is simple and is detailed in this paper.

## 6 Conclusion

The constructs that are supported in a programming language often depend on the compiler support available to leverage those constructs for efficient implementations, the frequency of their usage by programmers, and their intuitiveness. Non-deterministic constructs have been a topic of much discussion, and language designers may benefit through more clarity on both optimization leverage and usage metrics. Through our work, we produce evidence showing that (a) programmers are using unsequenced expressions with side effects freely and correctly in their C code even when they may not be aware of the potential performance benefits; and (b) the optimization opportunity presented by leveraging the extra information provided through such programming patterns can potentially be significant.

## References


[1] 2019. Discussion on restrict support in LLVM. https://lists.llvm.org/pipermail/llvm-dev/2019-October/135672.html.

[2] 2019. Polybench/C. https://sourceforge.net/projects/polybench/.

[3] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

[4] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43, 3 (01 Oct 2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

[5] Gabriel Dos Reis, Herb Sutter, and Jonathan Caves. 2016. Refining Expression Evaluation Order for Idiomatic C++ (Revision 2). (02 2016).

[6] Chucky Ellison and Grigore Rosu. 2012. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. ACM, New York, NY, USA, 533–544. https://doi.org/10.1145/2103656.2103719

[7] Yuri Gurevich and James K. Huggins. 1993. The Semantics of the C Programming Language. In *Selected Papers from the Workshop on Computer Science Logic (CSL '92)*. Springer-Verlag, London, UK, UK, 274–308. http://dl.acm.org/citation.cfm?id=647842.736414

[8] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 336–345. https://doi.org/10.1145/2737924.2737979

[9] Michael Hind. 2001. Pointer Analysis: Haven'T We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, Utah, USA) *(PASTE '01)*. ACM, New York, NY, USA, 54–61. https://doi.org/10.1145/379605.379665

[10] Robbert Krebbers. 2014. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 101–112. https://doi.org/10.1145/2535838.2535878

[11] Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 42–54. http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf

[12] Markus Mock. 2004. Why Programmer-specified Aliasing is a Bad Idea. (01 2004).

[13] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/2491956.2491967

[14] M Norrish. 1998. Formalized in HOL. PhD Thes. University of Cambridge. *Computer Laboratory* (1998).

[15] Michael Norrish. 1999. Deterministic Expressions in C. In *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*. 147–161. https://doi.org/10.1007/3-540-49099-X_10

[16] Michael Norrish. 2002. An abstract dynamic semantics for C. (10 2002).

[17] Nikolaos S. Papaspyrou. 2001. Denotational semantics of ANSI C. *Computer Standards & Interfaces* 23, 3 (2001), 169 – 185. https://doi.org/10.1016/S0920-5489(01)00059-9

[18] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 260–275. https://doi.org/10.1145/2517349.2522728