Optimization Remarks

Helping the Compiler Generate Better Code

OFEK SHILON

Ofek Shilon

Developer
@Istra Research

ofekshilon@gmail.com

OfekShilon

Takeaways

- Clang optimization remarks:
  Shed light on optimizations that were tried but failed.

- Learn how to:
  - get them,
  - understand (some of) them,
  - solve (some of) them, get better optimizations.

Part 1: Getting Them

-Rpass

-Rpass sample output

```
Parser/pegen_errors.c:142:9: remark: load of type %struct._object* not eliminated [-Rpas
s-missed=gvn]
Parser/pegen_errors.c:53:8: remark: RAISE_ERROR_KNOWN_LOCATION has uninlinable pattern (
varargs) and cost is not fully computed [-Rpass-missed=inline-cost]
       RAISE_ERROR_KNOWN_LOCATION(p, PyExc_SyntaxError,
       ^
Parser/pegen_errors.c:53:8: remark: 'RAISE_ERROR_KNOWN_LOCATION' not inlined into '_PyPe
gen_tokenize_full_source_to_check_for_errors' because it should never be inlined (cost=n
ever): varargs [-Rpass-missed=inline]
Parser/pegen_errors.c:171:37: remark: failed to move load with loop-invariant address be
cause the loop may invalidate its value [-Rpass-missed=licm]
        switch (_PyTokenizer_Get(p->tok, &start, &end)) {
                                    ^
Parser/pegen_errors.c:171:37: remark: failed to move load with loop-invariant address be
cause the loop may invalidate its value [-Rpass-missed=licm]
Parser/pegen_errors.c:163:31: remark: load of type %struct.Token* not eliminated [-Rpass
-missed=gvn]
    Token *current_token = p->known_err_token != NULL ? p->known_err_token : p->tokens[p
->fill - 1];
                               ^
Parser/pegen_errors.c:163:81: remark: load of type %struct.Token** not eliminated [-Rpas
s-missed=gvn]
    Token *current_token = p->known_err_token != NULL ? p->known_err_token : p->tokens[p
->fill - 1];
                                                                                ^
```

-Rpass

llvm-opt-report

llvm-opt-report

- https://reviews.llvm.org/D25262
- https://github.com/llvm/llvm-project/tree/main/llvm/tools/llvm-opt-report

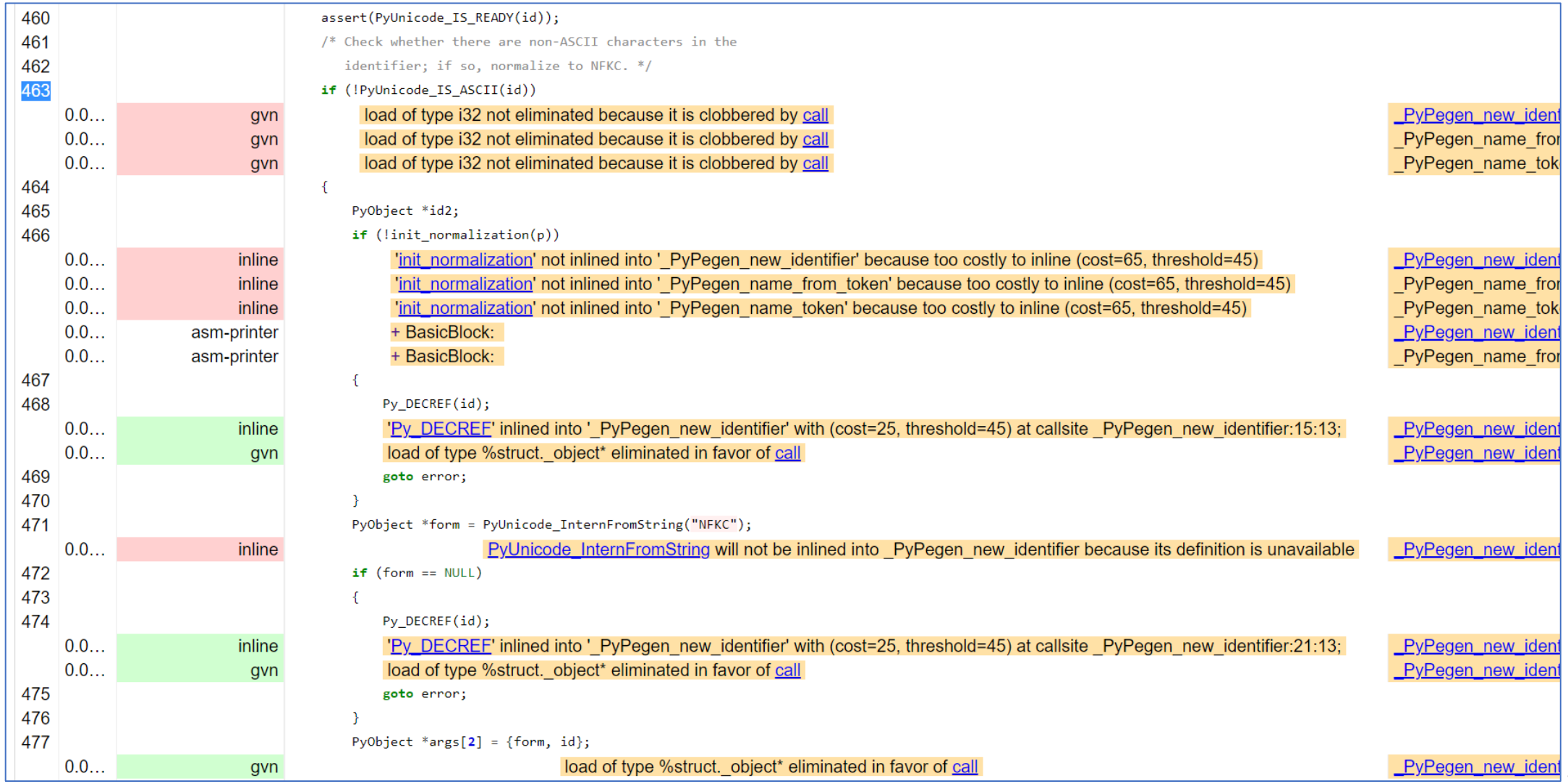
```
< /tmp/v.c
 2           | void bar();
 3           | void foo() { bar(); }
 4           |
 5           | void Test(int *res, int *c, int *d, int *p, int n) {
 6           |   int i;
 7           |
 8           | #pragma clang loop vectorize(assume_safety)
 9      V4,2 |   for (i = 0; i < 1600; i++) {
10           |     res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
11           |   }
12           |
13  U16      |   for (i = 0; i < 16; i++) {
14           |     res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
15           |   }
16           |
17 I         |   foo();
18           |
19           |   foo(); bar(); foo();
   I         |   ^
   I         |                 ^
20           | }
```

-Rpass

llvm-opt-report

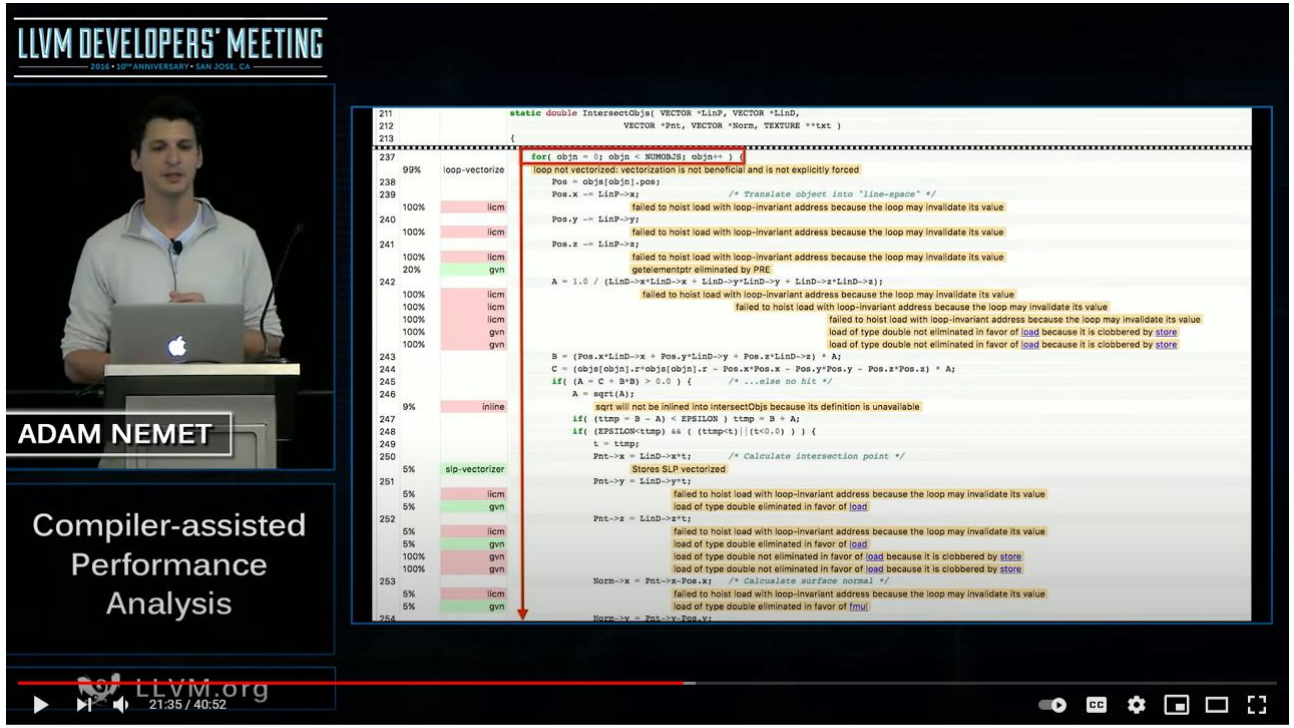
opt-viewer

opt-viewer sample output



opt-viewer

- 2016 work led by Adam Nemet (Apple) https://www.youtube.com/watch?v=qq0q1hfzidg
- Part of LLVM master: https://github.com/llvm/llvm-project/tree/main/llvm/tools/opt-viewer
- Downloadable via deb pkg: llvm-14-tools



opt-viewer Usage

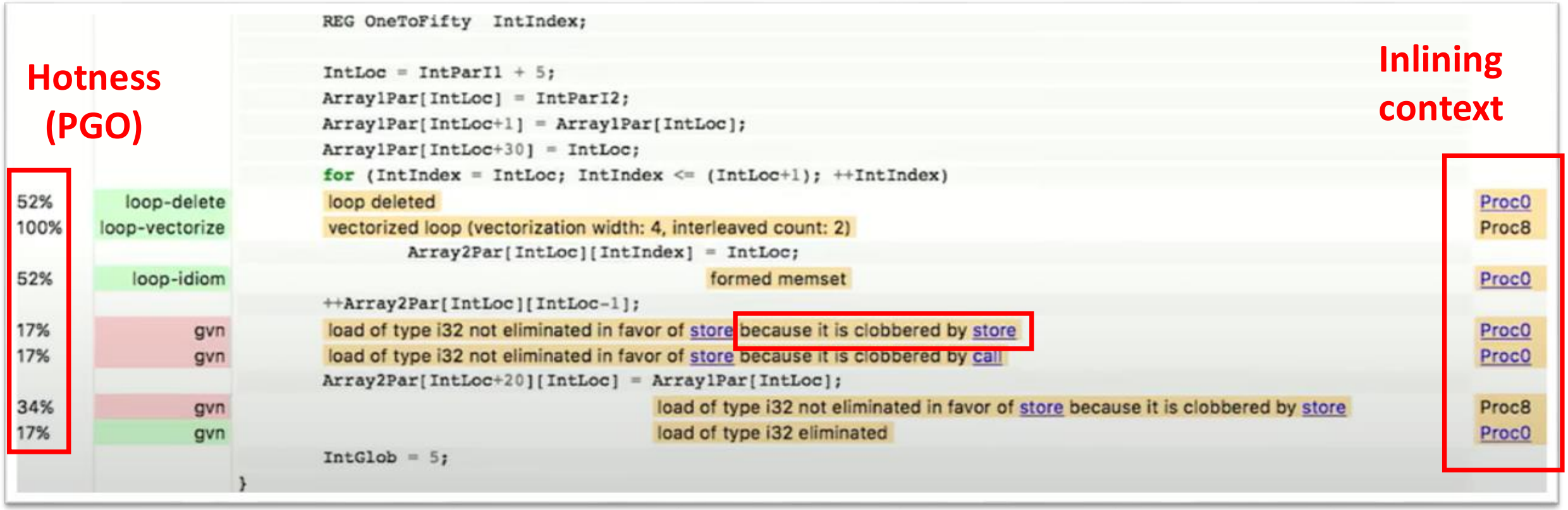
- Build with an extra clang switch:
  `-fsave-optimization-record`
  `*.opt.yaml` files are generated, by default in the obj folder.

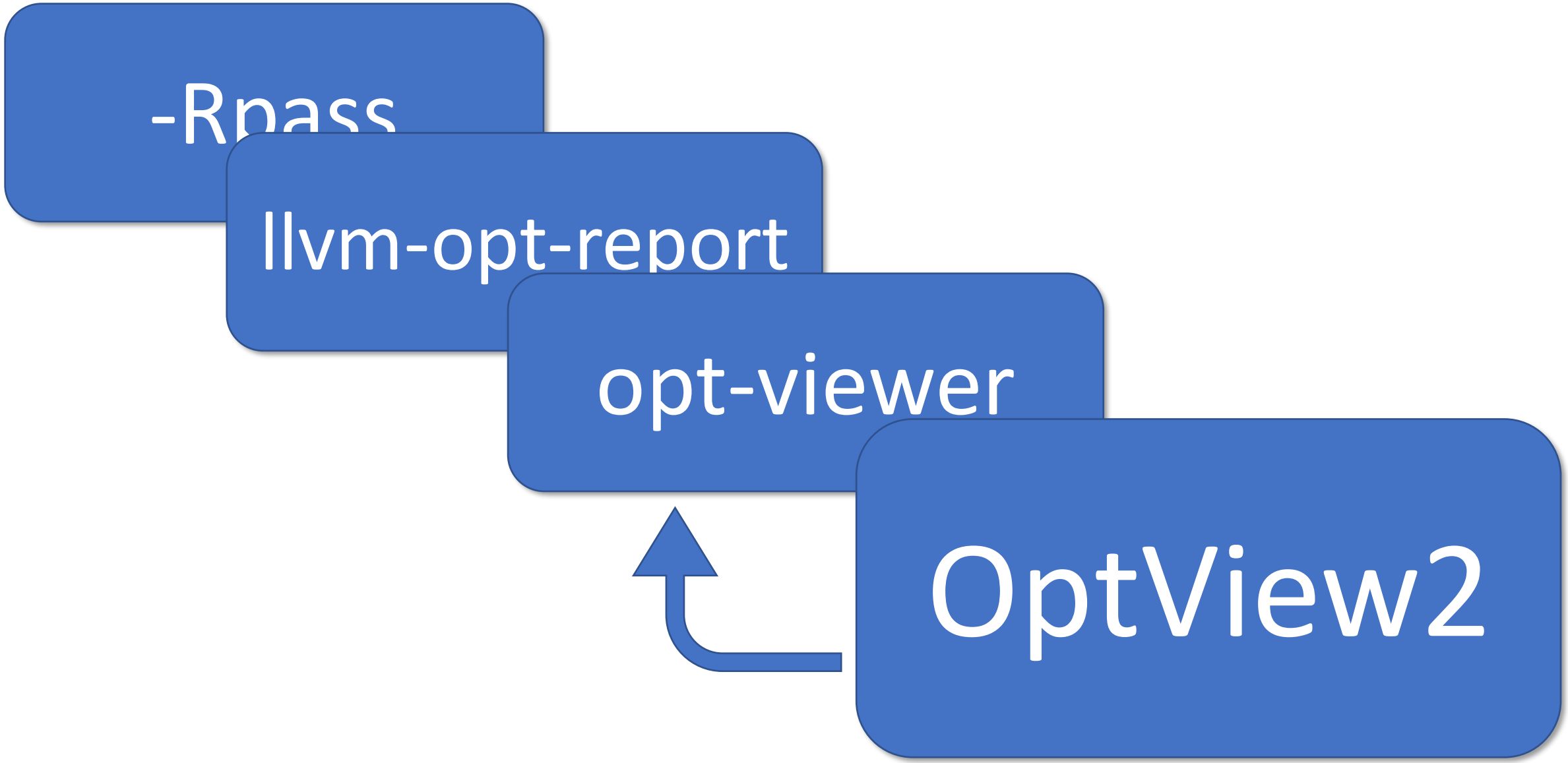
- Generate htmls:

```
$ opt-viewer.py
      --output-dir <htmls folder>
      --source-dir <repo>
      <yamls folder>
```

```
...
--- !Passed
Pass:            inline
Name:            Inlined
DebugLoc:        { File: '/usr/bin/../lib/gcc/
                   Line: 147, Column: 16 }
Function:        _ZNSt14pointer_traitsIPKcE10p
Args:
  - Callee:          _ZSt9addressofIKcEPT_RS1_
    DebugLoc:        { File: '/usr/bin/../lib/
                       Line: 139, Column: 0 }
  - String:          ' inlined into '
  - Caller:          _ZNSt14pointer_traitsIPKc
    DebugLoc:        { File: '/usr/bin/../lib/
                       Line: 147, Column: 0 }
  - String:          ' with '
  - String:          '(cost='
```

opt-viewer additions over -Rpass




-Rpass
llvm-opt-report
opt-viewer
OptView2

opt-viewer is great (seriously), but

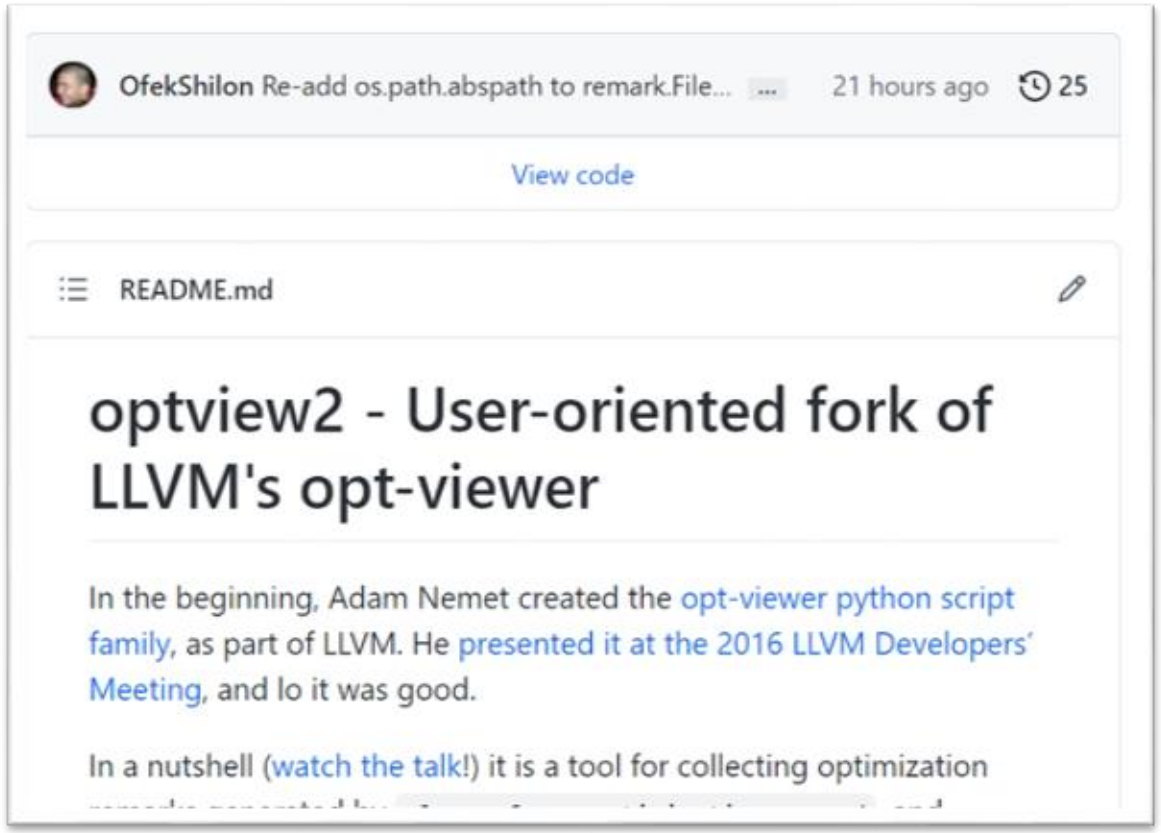
- Heavy
  - High I/O
  - High memory
  - >1G htmls
- Designed (and presented) for compiler authors
  - Mostly non actionable to developers

Improvements in OptView2

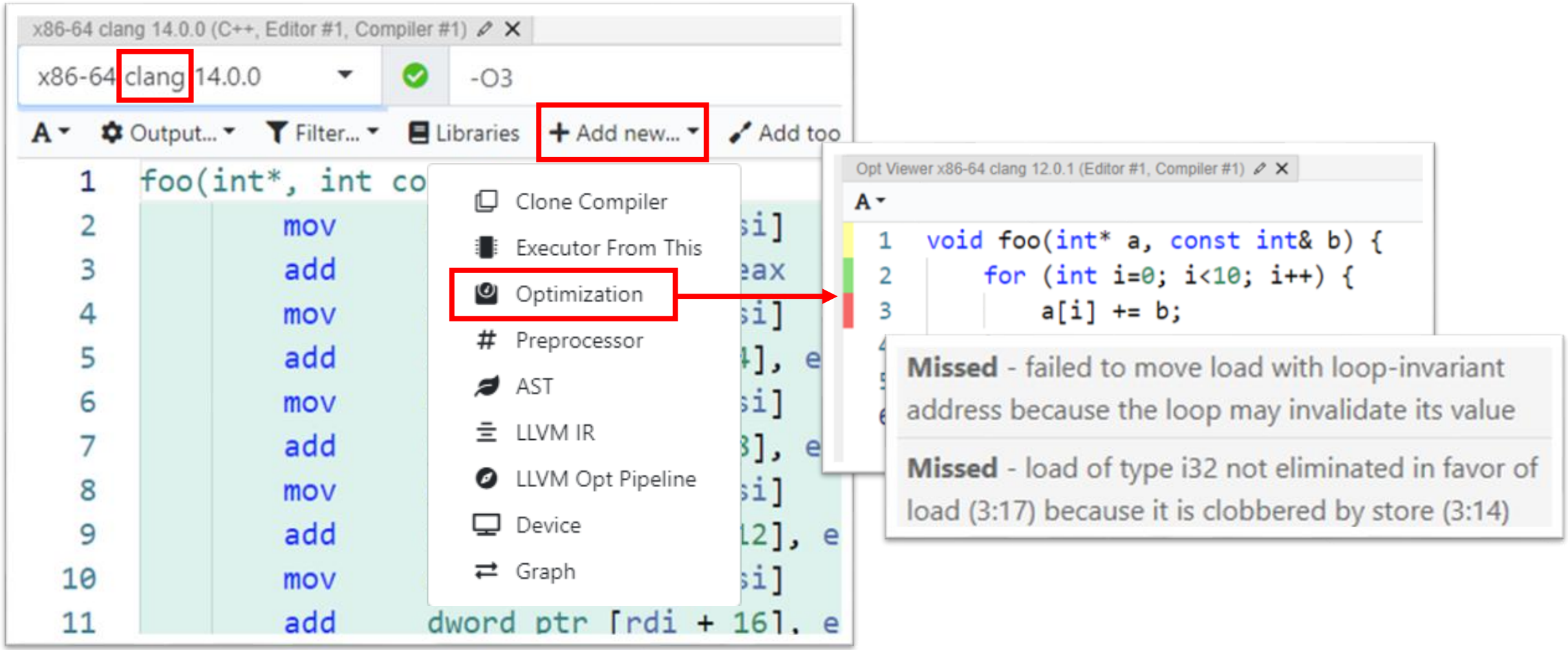
- Denoise:
  - Collect only optimization *failures* (by default)
  - Exclude system headers (by default)
  - Remove duplicities,
  - Filter remark types via config file/command line
- Optional split-to-subfolders
- Display column info (location within line)
- Sortable, resizable & pageable index
- ...

OptView2 – coming to trunk

- https://github.com/OfekShilon/optview2



View in compiler explorer!



Part 2: Understanding and Using Them

1. Inlining

https://ofekshilon.github.io/optview2-opencv/core/modules_core_include_opencv2_core_dualquaternion.inl.hpp.html#L80

```
    return createFromQuat(r, trans * r * T(0.5));
}

  • cv::Mat::~Mat will not be inlined into cv::DualQuat<double>::createFromMat because its definition is unavailable

template <typename T>
DualQuat<T> DualQuat<T>::createFromAffine3(const Affine3<T> &R)
{
    return createFromMat(R.matrix);
           ^
      • 'cv::DualQuat<double>::createFromMat' not inlined into 'cv::DualQuat<double>::createFromAffine3' because too
      (cost=645, threshold=625)
      • 'cv::DualQuat<double>::createFromMat' not inlined into 'opencv_test::(anonymous
      namespace)::DualQuatTest_constructor_Test::Body' because too costly to inline (cost=645, threshold=625)
```

2. "Clobbered by store"

https://godbolt.org/z/T7h4nK3G7

```
1  void foo(int* a, const int& b) {
2      for (int i=0; i<10; i++) {
3          a[i] += b;
4      }
5  }
```

```
1   foo(int*, int const&):
2           movl    (%rsi), %eax
3           addl    %eax, (%rdi)
4           movl    (%rsi), %eax
5           addl    %eax, 4(%rdi)
6           movl    (%rsi), %eax
7           addl    %eax, 8(%rdi)
8           movl    (%rsi), %eax
9           addl    %eax, 12(%rdi)
10          movl    (%rsi), %eax
11          addl    %eax, 16(%rdi)
12          movl    (%rsi), %eax
13          addl    %eax, 20(%rdi)
14          movl    (%rsi), %eax
15          addl    %eax, 24(%rdi)
            (%rsi), %eax
            %eax, 28(%rdi)
            (%rsi), %eax
            %eax, 32(%rdi)
```

Opt Viewer x86-64 clang 12.0.1 (Editor #1, Compiler #1)

A

```
1  void foo(int* a, const int& b) {
2
3
4
5  }
```

Missed - load of type i32 not eliminated in favor of load (3:17) because it is clobbered by store (3:14)

2. "Clobbered by store"

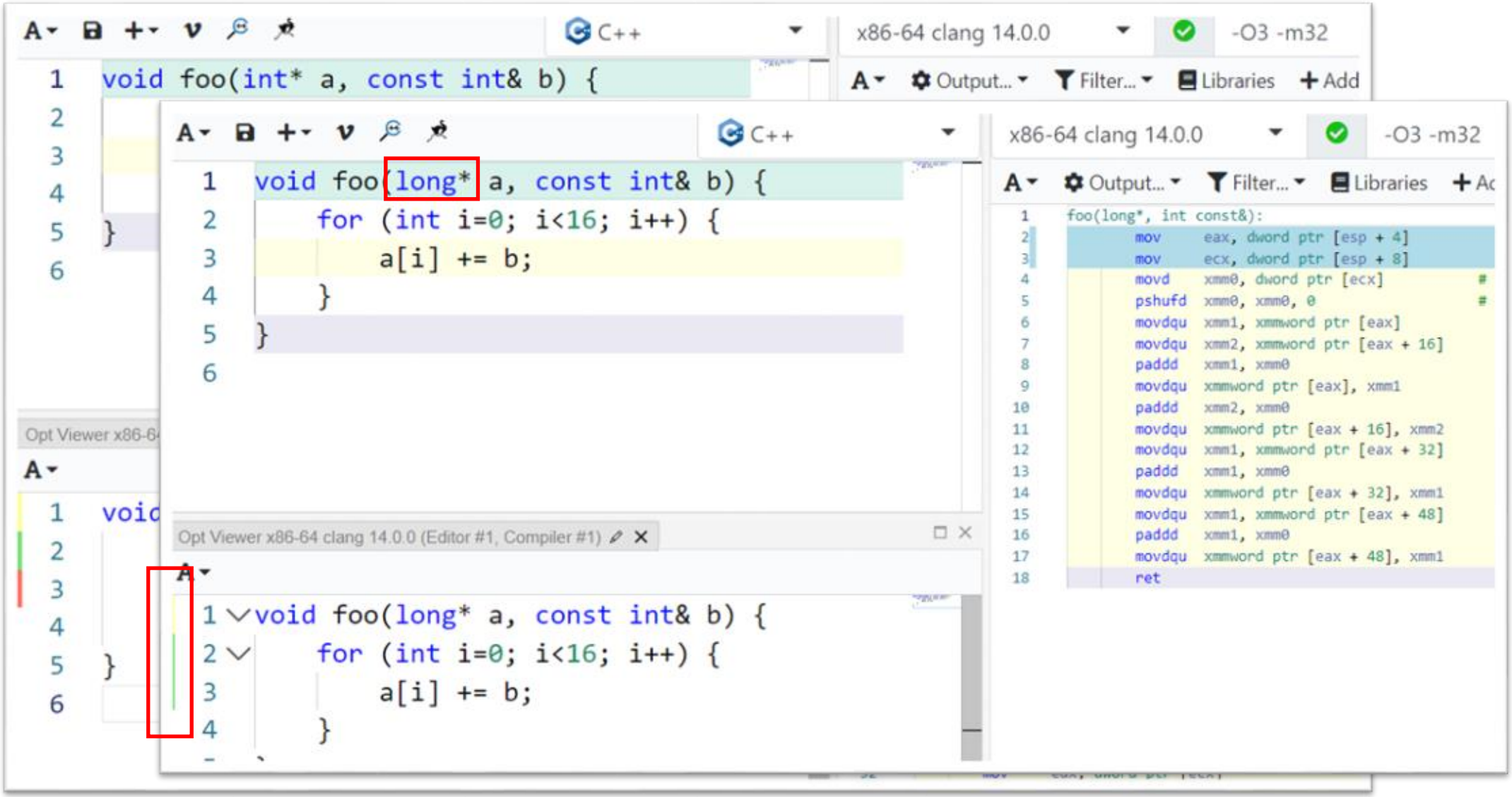
```
1  void foo(int* __restrict__ a,
2               const int& b) {
3      for (int i=0; i<10; i++) {
4          a[i] += b;
5      }
6  }
```

Opt Viewer x86-64 clang 12.0.1 (Editor #1, Compiler #1)

```
1  void foo(int* __restrict__ a,
2               const int& b) {
3      for (int i=0; i<10; i++) {
4          a[i] += b;
5      }
```

```
1   foo(int*, int const&):
2           movl    (%rsi), %eax
3           movd    %eax, %xmm0
4           pshufd  $0, %xmm0, %xmm0
5           movdqu  (%rdi), %xmm1
6           movdqu  16(%rdi), %xmm2
7           paddd   %xmm0, %xmm1
8           movdqu  %xmm1, (%rdi)
9           paddd   %xmm0, %xmm2
10          movdqu  %xmm2, 16(%rdi)
11          addl    %eax, 32(%rdi)
12          addl    %eax, 36(%rdi)
13          retq
```

2. "Clobbered by store"



2. "Clobbered by store"

- “Strict aliasing is **an assumption made by the compiler**, that objects of different types will never refer to the same memory location (i.e. alias each other.)”

  Mike Acton https://cellperformance.beyond3d.com/articles/2006/06/understanding-strict-aliasing.html

- Perhaps can be ‘weaponized’ to communicate non-aliasing to the compiler?
- In practice, compilers are struggling.
  - Example clang issue: https://github.com/llvm/llvm-project/issues/54646

3. “Clobbered by call”

https://godbolt.org/z/jG5jq7c9a

```
void somefunc(const int&);
int whateva();

void f(int i, int* res) {
    somefunc(i);
    i++;
    res[0] = whateva();
    i++;
    res[1] = whateva();
    i++;
    res[2] = whateva();
}
```

```
1  void somefunc(const int&);
2  int whateva();
3
4  void f(int i, int* res) {
5      somefunc(i);
6      i++;
7      res[0] = whateva();
8      i++;
9
10
11
12 }
```

```
1  f(int, int*):
2      pushq   %rbx
3      subq    $16, %rsp
4      movq    %rsi, %rbx
5      movl    %edi, 12(%rsp)
6      leaq    12(%rsp), %rdi
7      callq   somefunc(int const&)
8      incl    12(%rsp)
       callq   whateva()@PLT
       movl    %eax, (%rbx)
       incl    12(%rsp)
       callq   whateva()@PLT
       movl    %eax, 4(%rbx)
       incl    12(%rsp)
       callq   whateva()@PLT
       movl    %eax, 8(%rbx)
```

Missed - load of type i32 not eliminated in favor of store (8:6) because it is clobbered by call (9:14)

3. “Clobbered by call”

```
void somefunc(const int&) __attribute__((pure));
int whateva();

void f(int i, int* res) {
    somefunc(i);
    i++;
    res[0] = whateva();
    i++;
    res[1] = whateva();
    i++;
    res[2] = whateva();
}
```

```
1   f(int, int*):
2       pushq   %rbx
3       movq    %rsi, %rbx
4       callq   whateva()@PLT
5       movl    %eax, (%rbx)
6       callq   whateva()@PLT
7       movl    %eax, 4(%rbx)
8       callq   whateva()@PLT
9       movl    %eax, 8(%rbx)
10      popq    %rbx
11      retq
```

- Cheating?.. pure + returns void `somefunc()` – does nothing, removed entirely.
- If returned non-void – wouldn’t work (clang issue: https://github.com/llvm/llvm-project/issues/53102)

3. “Clobbered by call”

```
void somefunc(const int&);
int whateva() __attribute__((const));

void f(int i, int* res) {
    somefunc(i);
    i++;
    res[0] = whateva();
    i++;
    res[1] = whateva();
    i++;
    res[2] = whateva();
}
```

```
1   f(int, int*):
2           pushq   %rbx
3           subq    $16, %rsp
4           movq    %rsi, %rbx
5           movl    %edi, 12(%rsp)
6           leaq    12(%rsp), %rdi
7           callq   somefunc(int const&)
8           callq   whateva()@PLT
9           movl    %eax, (%rbx)
10          movl    %eax, 4(%rbx)
11          movl    %eax, 8(%rbx)
12          addq    $16, %rsp
13          popq    %rbx
14          retq
```

- Whateva() called only once, result copied to 2 other places

3. “Clobbered by call”

```
void somefunc(const int& __attribute__((noescape)));
int whateva();

void f(int i, int* res) {
    somefunc(i);
    i++;
    res[0] = whateva();
    i++;
    res[1] = whateva();
    i++;
    res[2] = whateva();
}
```

```
1   f(int, int*):
2           pushq   %rbx
3           subq    $16, %rsp
4           movq    %rsi, %rbx
5           movl    %edi, 12(%rsp)
6           leaq    12(%rsp), %rdi
7           callq   somefunc(int const&
8           callq   whateva()@PLT
9           movl    %eax, (%rbx)
10          callq   whateva()@PLT
11          movl    %eax, 4(%rbx)
12          callq   whateva()@PLT
13          movl    %eax, 8(%rbx)
14          addq    $16, %rsp
15          popq    %rbx
16          retq
```

3. “Clobbered by call”

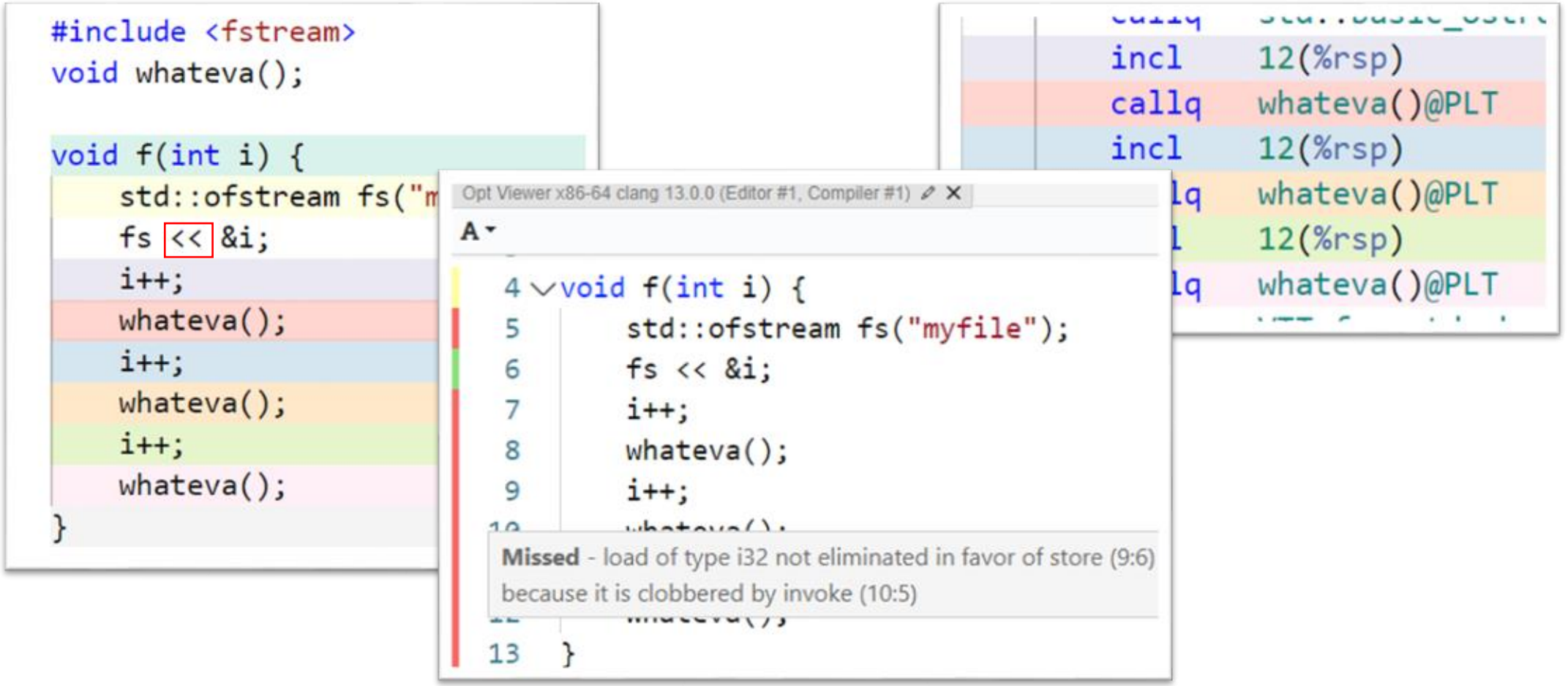
```
void somefunc(const int&);
int whateva();

void f(int i, int* res) {
    somefunc(+i);
    i++;
    res[0] = whateva();
    i++;
    res[1] = whateva();
    i++;
    res[2] = whateva();
}
```

```
1   f(int, int*):
2           pushq   %rbx
3           subq    $16, %rsp
4           movq    %rsi, %rbx
5           movl    %edi, 12(%rsp)
6           leaq    12(%rsp), %rdi
7           callq   somefunc(int const&
8           callq   whateva()@PLT
9           movl    %eax, (%rbx)
10          callq   whateva()@PLT
11          movl    %eax, 4(%rbx)
12          callq   whateva()@PLT
13          movl    %eax, 8(%rbx)
14          addq    $16, %rsp
15          popq    %rbx
16          retq
```

3. “Clobbered by call”

Sometimes the offending call is standard! https://godbolt.org/z/81319zq1E



4. “Failed to move load with loop invariant address”

https://godbolt.org/z/YGc83TMnj

```
class C {
    bool m_cond;
    void method1();
};
void f();

void C::method1() {
    auto cond = m_cond;
    for (int i=0; i<5; ++i)
        if (cond)
            f();
}
```

```
C::method1():
        push    rax
        cmp     byte ptr [rdi], 0
        je      .LBB0_1
        call    f()@PLT
        call    f()@PLT
        call    f()@PLT
        call    f()@PLT
        pop     rax
        jmp     f()@PLT
.LBB0_1:
        pop     rax
        ret
```

```
C::method1():
        push    rbx
        cmp     by
        je      .L
        mov     rb
        call    f(
        cmp     by
        je      .L
        call    f(
        cmp     by
        je      .L
        call    f(
        cmp     by
        je      .L
        call    f(
        cmp     byte ptr [rbx], 0
        pop     rbx
        ret
```

- failed to move load with loop-invariant address because
p may invalidate its value

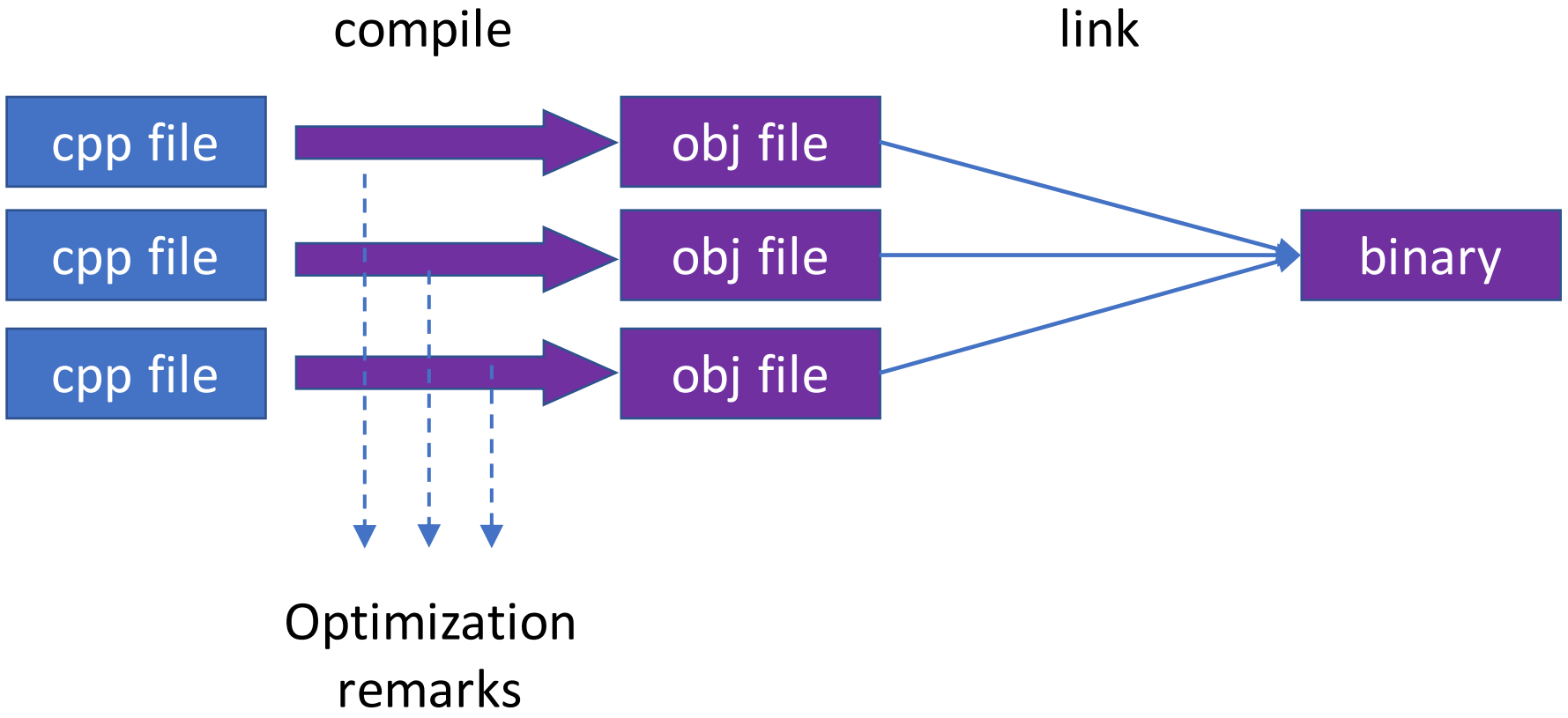
Cheat Sheet

Symptom	Probable cause	Action
Inlining Failure		Add header / forceinline / increase threshold
"Clobbered by store"	Aliasing	restrict / force type diff
"Clobbered by load"	Escape	Attributes pure / const / noescape (typically *before* the remark site)
"Failed to move load loop invariant"	Escape	All the above + copy to local
*	Don’t understand?	Reduce to bare minimum in godbolt. Might be a compiler limitation.

Part 3: Beyond Classical Clang Toolchain

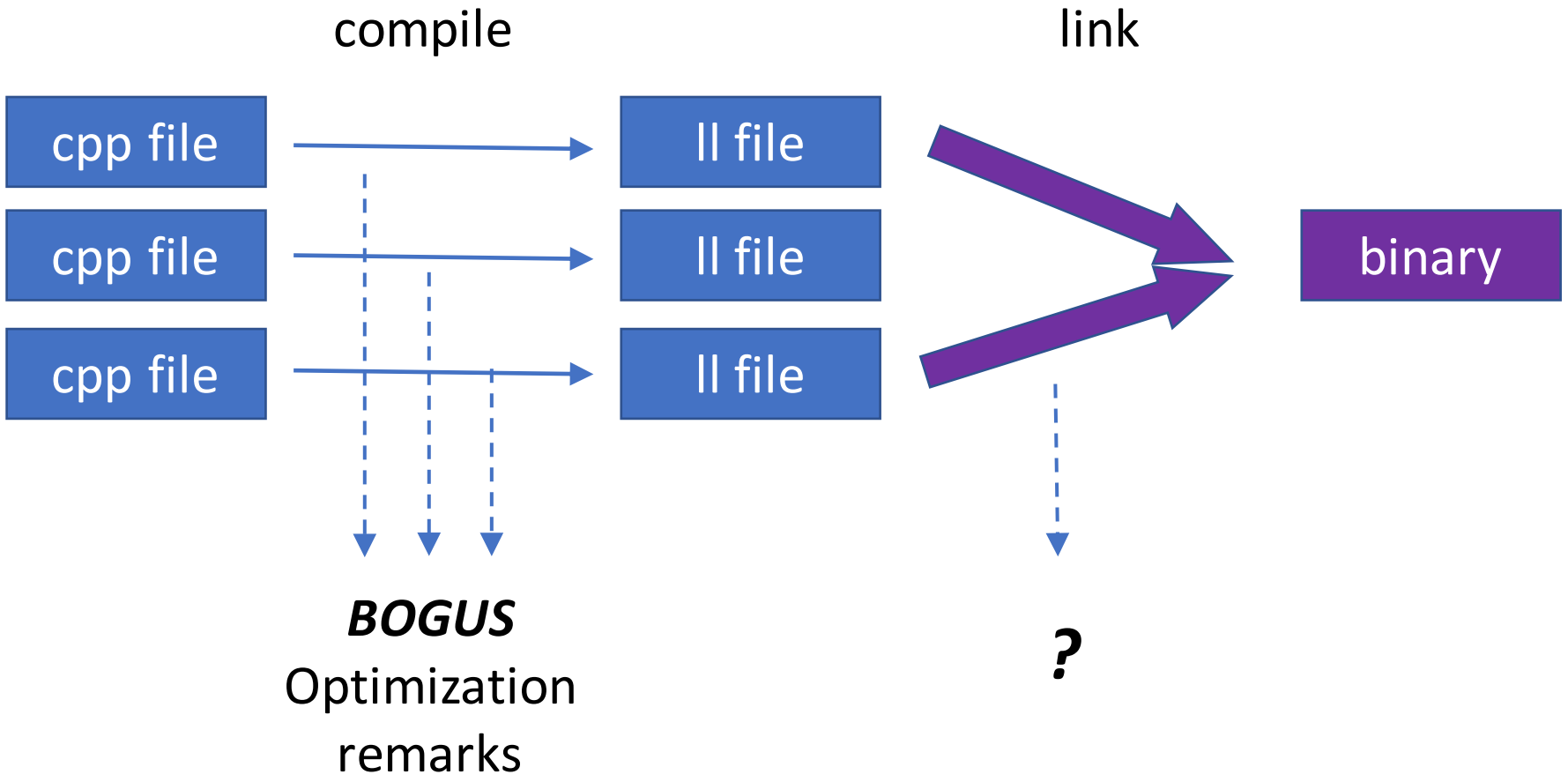
Opt remarks with LTO

Regular build:



Opt remarks with LTO

LTO build:



Opt remarks with LTO

- To get remarks from linker:
  - Build with LTO, use –v to dump the list of LL files generated
  - Form a binary:

```
$ llvm-lto -lto-pass-remarks-output=<yaml outputpath>
  -j=10 -O=3 <obj files list>
```

- Creates a *single* huge yaml.
- No parallelism in creation or consumption by opt-viewer.
- **Hard to get meaningful results for a large project.**

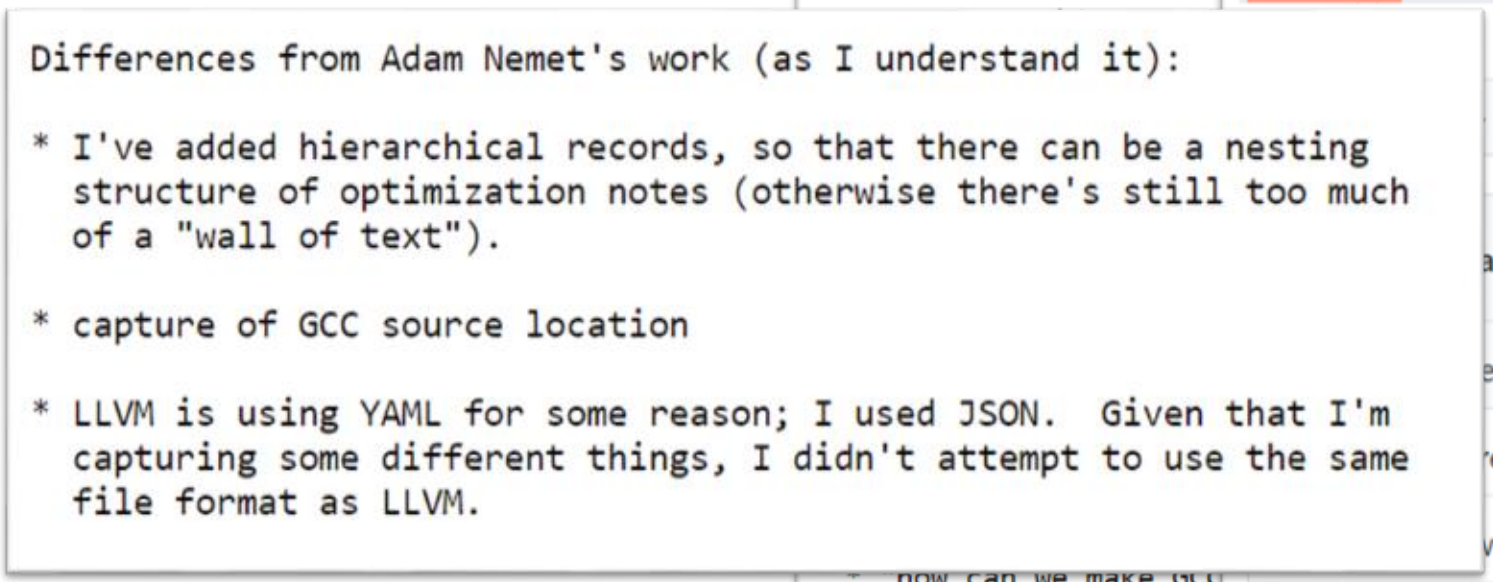
Opt remarks with LTO

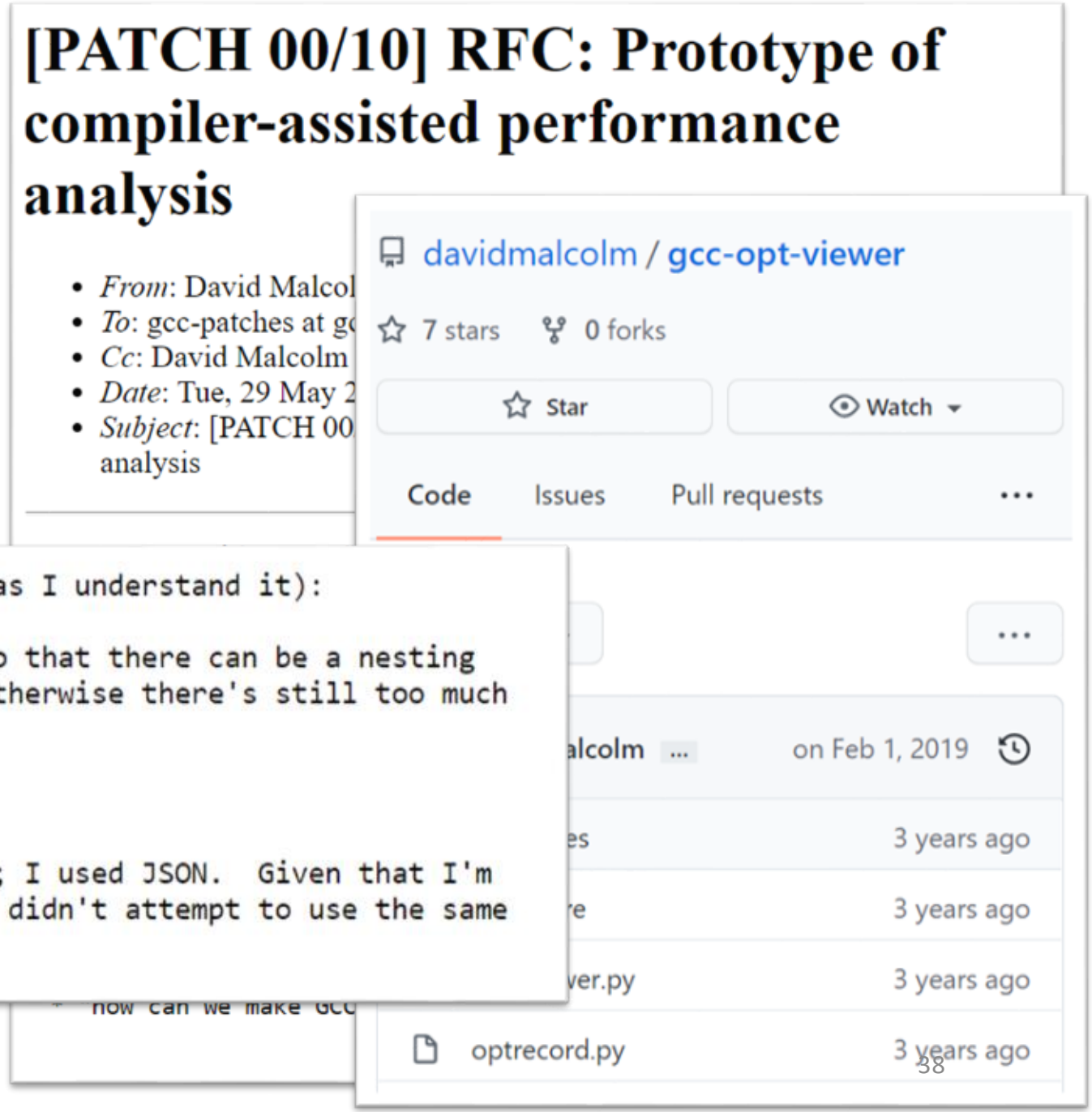
- Inlining – non-issue.
- Escape & Aliasing – still very much an issue.

- “inter-procedural analyses are often less precise … In LLVM, intra-procedural analyses are dominating in numbers and potential. “
  (Doerfert, Homerding, Finkel 2019)

GCC work

- https://gcc.gnu.org/legacy-ml/gcc-patches/2018-05/msg01675.html
- https://github.com/davidmalcolm/gcc-opt-viewer





GCC work

- https://dmalcolm.fedorapeople.org/gcc/2018-05-18/pgo-demo-test/pgo-demo-test/

GCC work

- Active only during 2018
- Still at prototype quality
  - Compilation might consume 10G+ RAM per single file
  - Python scripts often break
    - Opened two bugs, one solved in my fork

Decorations across compilers

clang	gcc	icc	msvc
__restrict	V	V	__restrict * __declspec(restrict) **
__attribute__((pure))	V	-	-
__attribute__((const))	V	V	__declspec(noalias)
__attribute__((noescape))	-	-	-

* Pertains also to locals

** Decorates a function return value

Decorations across compilers

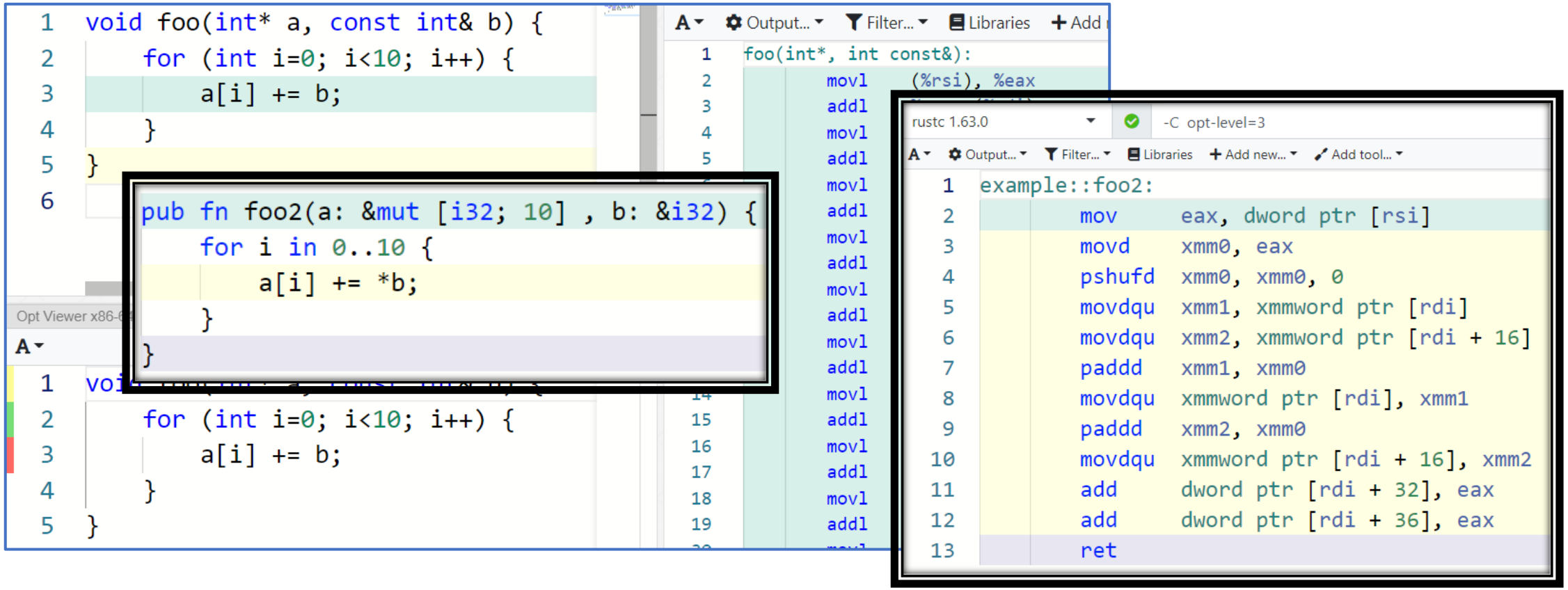
- `Hedley` (https://github.com/nemequ/Hedley) is a single header including cross-compiler wrappers like:

```
#if HEDLEY_HAS_ATTRIBUTE(noescape)
#  define HEDLEY_NO_ESCAPE __attribute__((__noescape__))
#else
#  define HEDLEY_NO_ESCAPE
#endif
```

- Known limitation: noalias
  - Check if still applicable: https://github.com/nemequ/hedley/issues/54)
  - Or use the fork: https://github.com/OfekShilon/hedley
- Can be used to find analogues in other compilers (Sun pragmas etc.)

Rust

The borrow-checker has some good news for alias analysis: https://godbolt.org/z/q9ox6n755



Carbon

Carries some good news about escape analysis:
https://www.foonathan.net/2022/07/carbon-calling-convention/

```
void somefunc(const int&);
int whateva();

void f(int i, int* res)
    somefunc(i);
    i++;
    res[0] = whateva();
    i++;
    res[1] = whateva();
    i++;
    res[2] = whateva();
}
```

```
fn somefunc(var i: i32);
fn whateva() -> i32;

fn f(var i: i32, var res: [i32; 3]) {
    somefunc(i);
    i = i+1;
    res[0] = whateva();
    i = i+1;
    res[1] = whateva();
    i = i+1;
    res[2] = whateva();
}
```

https://godbolt.org/z/Eo1jv97dW

Impact?

- Academic works
  - PETOSPA: …. Optimistic Static Program Annotations (Doerfert, Homerding, Finkel 2019)
    https://github.com/jdoerfert/PETOSPA/blob/master/ISC19.pdf
    - ~15%-20% speedup
  - ORAQL: Optimistic Responses to Alias Queries in LLVM (Hückelheim, Doerfert 2021)
    https://www.youtube.com/watch?v=7UVB5AFJM1w
    - No impact
  - HTO: … Optimization via Annotated Headers (Moses, Doerfert 2019)
    https://www.youtube.com/watch?v=elmio6AoyK0
    - ~50% of full LTO gains
- Personal experience: 6 µs -> 4.6 µs

Recommendations

- Concentrate on known bottlenecks,
- Invest when you
  - **work at sub-millisecond scale, or**
  - **in *very* tight loops.**

Final Musing

- The compiler *can* talk to you.
- You can learn to listen.
- And even answer.

- Sometimes.

Acknowledgements:

- Ilan Ben Hagai
- Oded Sharon
- Gal Falcon
- Roi Barkan
- Lior Solodkin

@OfekShilon

Optimization Remarks

Helping the Compiler Generate Better Code

OFEK SHILON

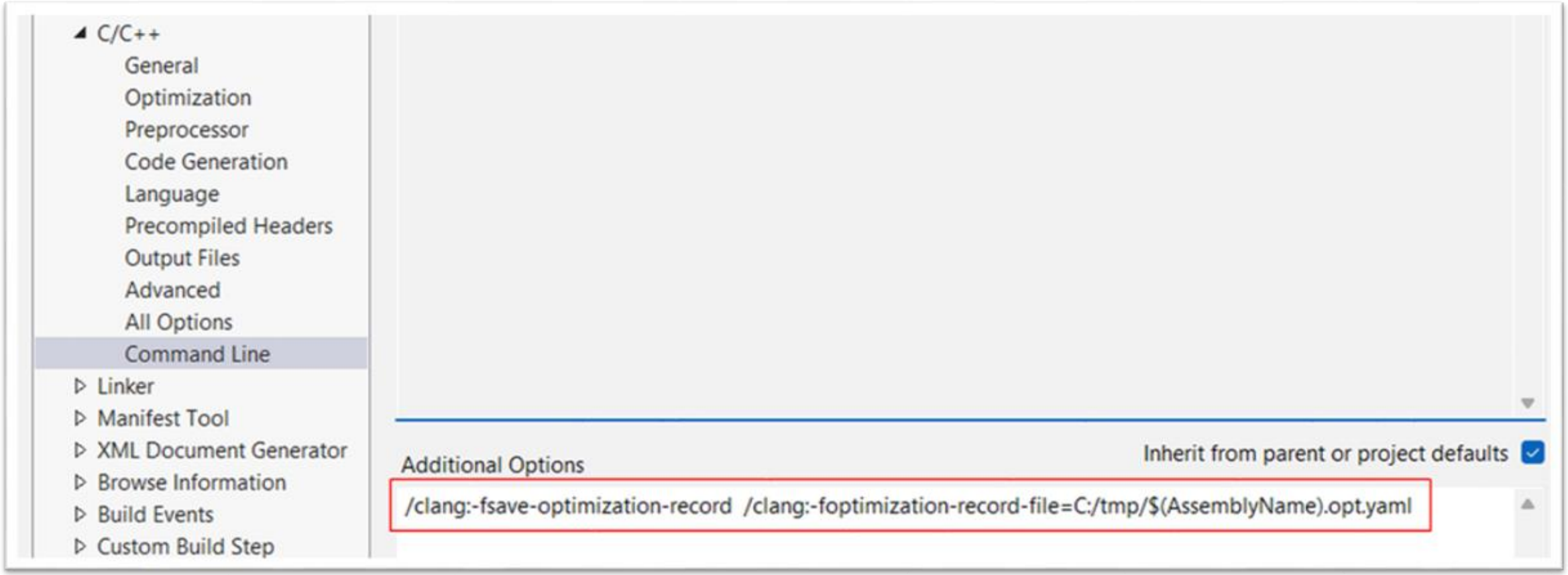
4. “Failed to move load with loop invariant address”

- Foreach or other <algorithm>s?
- In this toy example – identical code.
  - https://godbolt.org/z/jYWhG6zWc
- Occasionally different, not always better.

```
1   struct Wrapper1 { long int t; };
2   struct Wrapper2 { long int t; };
3   struct S { Wrapper1 a; Wrapper2 b; };
4
5   // Assignment optimized properly:
6   void f1(S& s1, S& s2 ) {
7       s1 = s2;
8   }
9
10  // Assignment not optimized due to bogus potential
11  // aliasing between a and b (see opt remarks):
12  void f2(S& s1, S& s2) {
13      s1.a = s2.a;
14      s1.b = s2.b;
15  }
16
17  // Assignment optimized properly:
18  void f3(S& s1, S& s2) {
19      s1.a.t = s2.a.t;
20      s1.b.t = s2.b.t;
21  }
```

```
f1(S&, S&):                                  # @
        movups  xmm0, xmmword ptr [rsi]
        movups  xmmword ptr [rdi], xmm0
        ret
f2(S&, S&):                                  # @
        mov     rax, qword ptr [rsi]
        mov     qword ptr [rdi], rax
        mov     rax, qword ptr [rsi + 8]
        mov     qword ptr [rdi + 8], rax
        ret
f3(S&, S&):                                  # @
        movups  xmm0, xmmword ptr [rsi]
        movups  xmmword ptr [rdi], xmm0
        ret
```

Clang-cl usage



4. “Failed to move load with loop invariant address”

- https://godbolt.org/z/jv3sa7cbs

```
1   class C {
2       int m_val;
3       void method1(int* a);
4   };
5   void f();
6   void C::method1(int* a) {
7       for(int i=0; i<5; ++i) {
8           if (m_val > a[i])
9               f();
10      }
11  }
12
```

Const method that modifies members

```
struct C {
    int  m_i;
    int* m_p = &m_i;
    void constMethod() const { ++(*m_p); };  // m_i modified
};
```