Optimization Remarks: Helping the Compiler Generate Better Code

Ofek Shilon

Compiler-Explorer maintainer

& Dev @Speedata

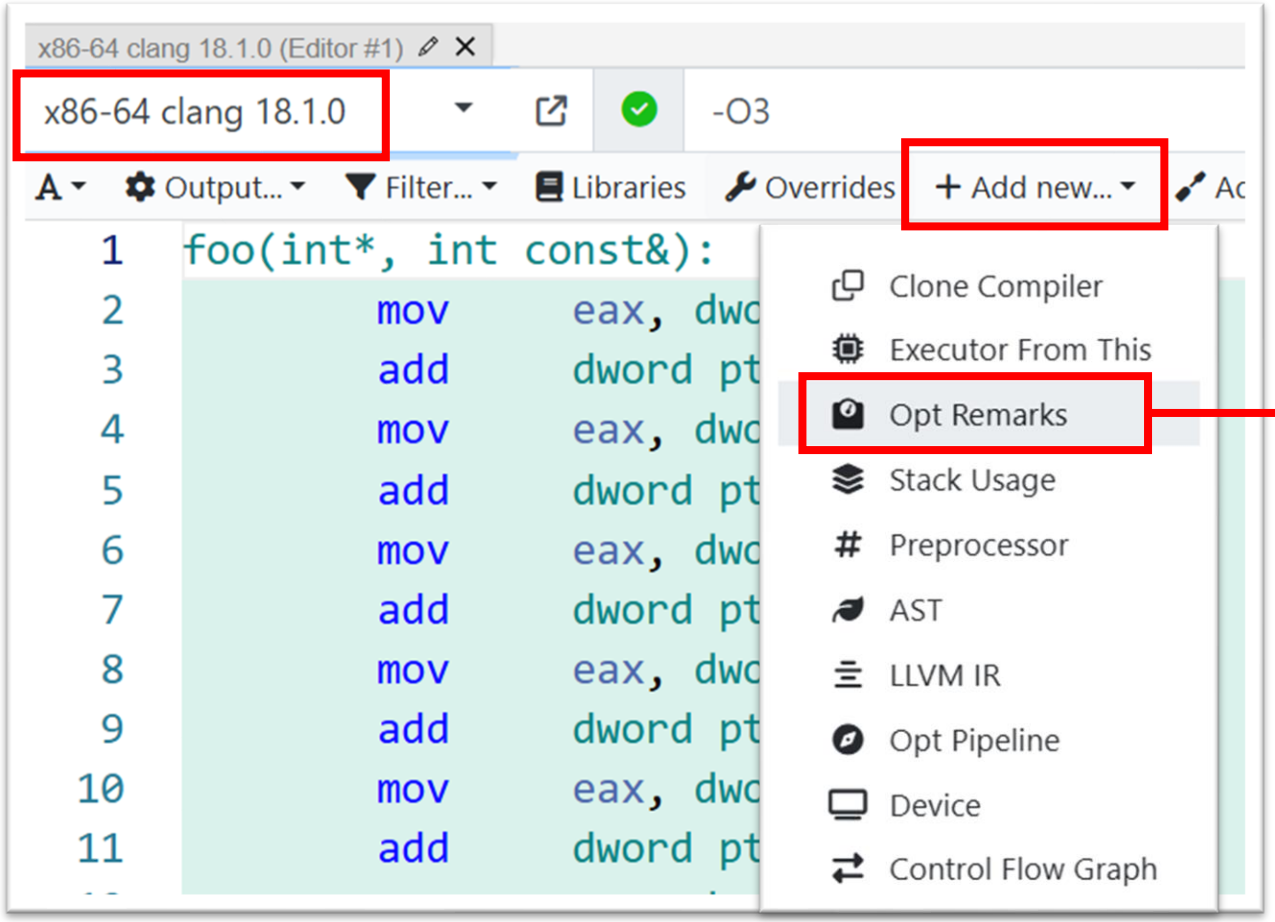
Takeaways

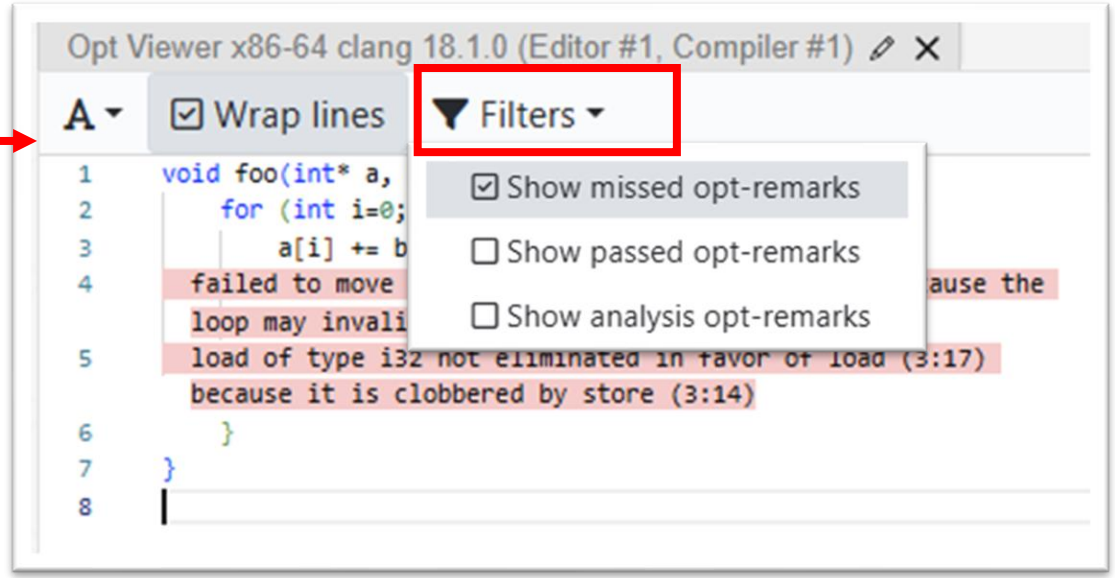
- Clang optimization remarks:
  - Shed light on various optimizations attempts, and reasons why some failed.

- Learn how to:
  - Interpret (some of) them,
  - solve (some of) them, get better optimizations.
  - obtain them for your own project

Part 1:
Understanding
and Using Opt-Remarks

View in compiler-explorer





1. Inlining

https://compiler-explorer.com/z/K1fK4YYoM

```
#include <fstream>
void whateva();

void f(int i) {
    std::ofstream fs("myfile");
    whateva();
}
```

```
#include <fstream>
void whateva();

void f(int i) {
  2 virtual registers copies 1.907349e-06 total copies cost generated in function
    std::ofstream fs("myfile");
  'std::basic_ofstream<char, std::char_traits<char>>::basic_ofstream (800:0)' not
  inlined into 'f (4:0)' because too costly to inline (cost=390, threshold=225)
    whateva();
  whateva will not be inlined into f (4:0) because its definition is unavailable
  'std::basic_ofstream<char, std::char_traits<char>>::~basic_ofstream (870:0)' not
  inlined into 'f (4:0)' because too costly to inline (cost=250, threshold=225)
  'std::basic_ofstream<char, std::char_traits<char>>::~basic_ofstream (870:0)' not
  inlined into 'f (4:0)' because too costly to inline (cost=65, threshold=45)
}
```

2. "Clobbered by store"

https://compiler-explorer.com/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)
            ovl     (%rsi), %eax
            dl      %eax, 20(%rdi)
            ovl     (%rsi), %eax
            dl      %eax, 24(%rdi)
            ovl     (%rsi), %eax
            dl      %eax, 28(%rdi)
            ovl     (%rsi), %eax
            dl      %eax, 32(%rdi)
```

```
void foo(int* a, const int& b) {
    for (int i=0; i<10; i++) {
        a[i] += b;
  failed to move load with loop-invariant address because the loop may invalidate its value
  load of type i32 not eliminated in favor of load (3:17) because it is clobbered by store (3:14)
  load of type i32 not eliminated because it is clobbered by store (3:14)
    }
}
```

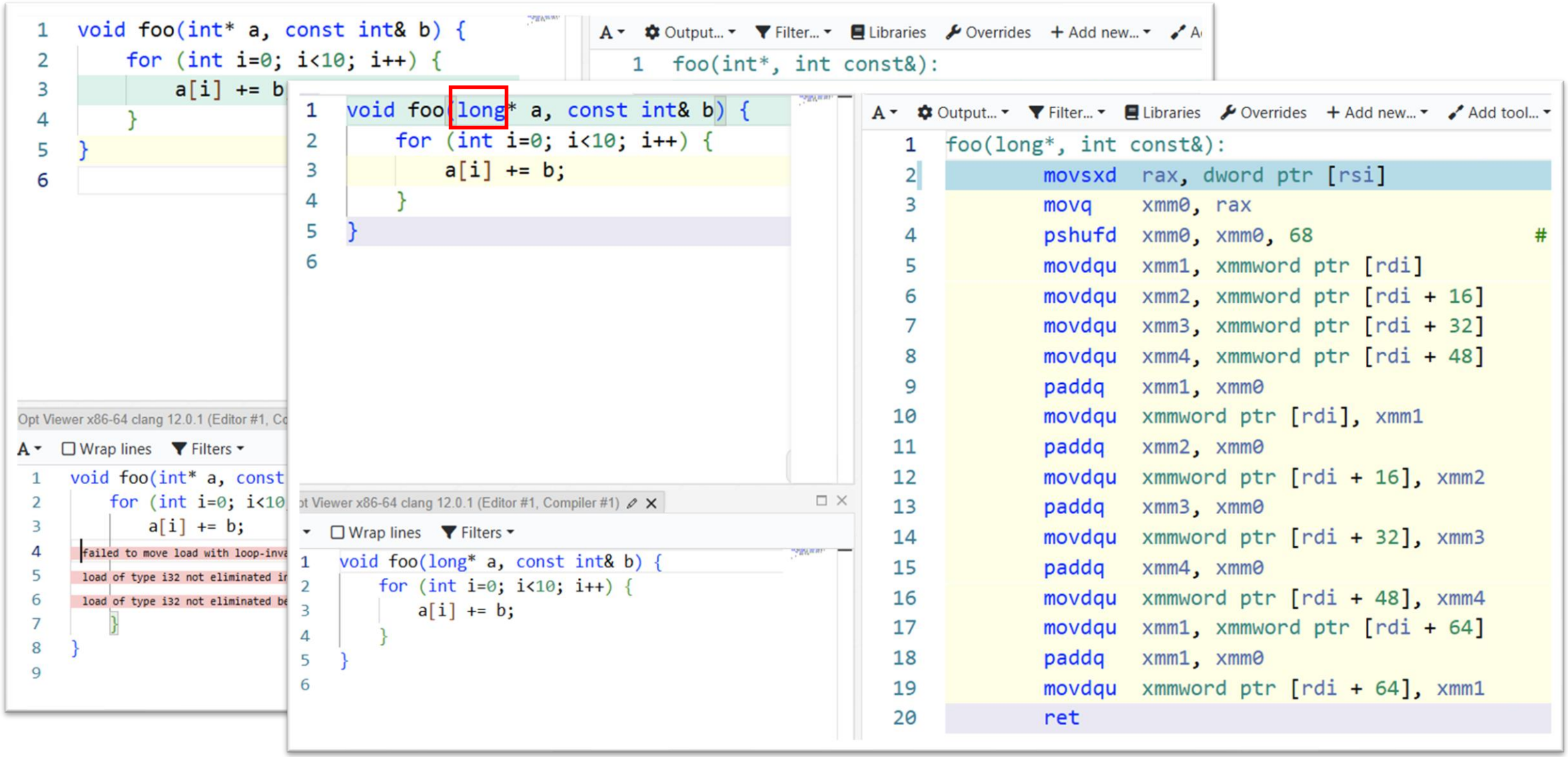
2. "Clobbered by store"

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

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

```
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

- Communicates non-aliasing to the compiler
  - strong-typedefs?
- In practice, compilers are struggling.

2. "Clobbered by store"

```
struct Wrapper { long int t; };
struct S { Wrapper a; Wrapper b; };

// Assignment vectorized
void f1(S& s1, const S& s2 ) {
    s1.a.t = s2.a.t;
    s1.b.t = s2.b.t;
}
// Assignment not vectorized due to aliasing
void f2(S& s1, const S& s2 ) {
    s1.a = s2.a;
    s1.b = s2.b;
}
```

```
f1(S&, S const&):
        movups  xmm0, xmmword ptr [rsi]
        movups  xmmword ptr [rdi], xmm0
        ret
f2(S&, S const&):
        mov     rax, qword ptr [rsi]
        mov     qword ptr [rdi], rax
```

```
1   struct Wrapper { long int t; };
2   struct S { Wrapper a; Wrapper b; };
3
4   // Assignment vectorized
5   void f1(S& s1, const S& s2 ) {
6       s1.a.t = s2.a.t;
7       s1.b.t = s2.b.t;
8   }
9   // Assignment not vectorized due to aliasing
10  void f2(S& s1, const S& s2 ) {
11      s1.a = s2.a;
12      s1.b = s2.b;
13    load of type i64 not eliminated because it is clobbered by store (11:10)
14  }
```

https://compiler-explorer.com/z/6oar5h4ao

Better alias analysis hidden in clang

- Experimental switch:

  `-Xclang -new-struct-path-tbaa`

- https://compiler-explorer.com/z/71Ed1cGcP

- Works in clang 15-17
  regressed in 18 ☹ https://github.com/llvm/llvm-project/issues/95661

Manual anti-aliasing

- A few past attempts to incorporate in the C++ standard
- Clang decided not to wait, and in v17 added:
  `__builtin_assume_separate_storage`

- Fixes the prev example: https://compiler-explorer.com/z/YPeofaYbE

```
void f2(S& s1, const S& s2 ) {
    __builtin_assume_separate_storage(&s1.a, &s2.b);
    s1.a = s2.a;
    s1.b = s2.b;
}
```

3. “Clobbered by call”

https://compiler-explorer.com/z/WvY6Knx9r

```
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();
}
```

```
f(int, int*):
        pushq   %rbx
        subq    $16, %rsp
        movq    %rsi, %rbx
        movl    %edi, 12(%rsp)
        leaq    12(%rsp), %rdi
        callq   somefunc(int const&)
        incl    12(%rsp)
        callq   whateva()@PLT
        movl    %eax, (%rbx)
```

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

void f(int i, int* res) {
    somefunc(i);
_Z8somefuncRKi will not be inlined into _Z1fiPi (4:0) because
its definition is unavailable
    i++;
load of type i32 not eliminated in favor of store because it is
clobbered by call (5:5)
    res[0] = whateva();
```

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
```

- Pure + returns void `somefunc()` = does nothing!
  Call removed entirely for gcc and clang<=17
- Clang 18 emits warning: 'pure' attribute on function returning 'void'; attribute ignored
- If returned non-void – doesn’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

```
#include <fstream>
void whateva();

void f(int i) {
    std::ofstream fs("myfile");
    fs << &i;
    i++;
    whateva();
    i++;
    whateva();
    i++;
    whateva();
}
```

```
        incl    12(%rsp)
        callq   whateva()@PLT
        incl    12(%rsp)
        callq   whateva()@PLT
        incl    12(%rsp)
```

```
#include <fstream>
void whateva();

void f(int i) {
    std::ofstream fs("myfile");
  _ZNSt14basic_ofstreamIcSt11char_traitsIcEEC1EPKcSt13_Ios_Openmode (798:0) not inlined into _Z
  inline (cost=390, threshold=250)
    fs << &i;
    i++;
  load of type i32 not eliminated in favor of store because it is clobbered by invoke (246:16)
    whateva();
```

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

https://compiler-explorer.com/z/bG7x6eKM4

```
class C {
    const bool m_cond = true;
    void method1() const;
    void method2() const;
};

void C::method1() const {
    for(int i=0; i<5; ++i) {
        if (m_cond)
          method2();
    }
}
```

```
        if (m_cond)
  failed to move load with loop-invariant address
  because the loop may invalidate its value
```

```
C::method1() const:                                    #
        pushq   %rbx
        cmpb    $0, (%rdi)
        je      .LBB0_5
        movq    %rdi, %rbx
        callq   C::method2() const@PLT
        cmpb    $0, (%rbx)
        je      .LBB0_5
        movq    %rbx, %rdi
        callq   C::method2() const@PLT
        cmpb    $0, (%rbx)
        je      .LBB0_5
        movq    %rbx, %rdi
        callq   C::method2() const@PLT
        cmpb    $0, (%rbx)
        je      .LBB0_5
        movq    %rbx, %rdi
        callq   C::method2() const@PLT
        cmpb    $0, (%rbx)
        je      .LBB0_5
        movq    %rbx, %rdi
        popq    %rbx
        jmp     C::method2() const@PLT
0_5:
        popq    %rbx
        retq
```

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

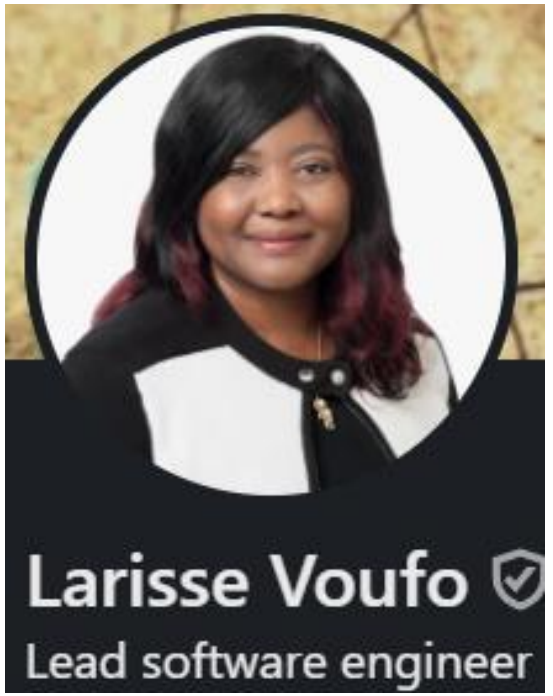
- **“Problem Statement**

  … the C++ ‘`const`’ keyword currently means nothing to LLVM,

  … constant variables with dynamic initializers, like C++ `const` objects, do not get the full benefit of optimizations currently supported in LLVM.”

- 2015 attempt to teach LLVM IR about constness, by Larisse Voufo

https://docs.google.com/document/d/112O-Q_XrbrU1I4P-oiLCN9u86Qg_BYBdcDsmh7Pn9Nw/edit#heading=h.trjqebywdvx4

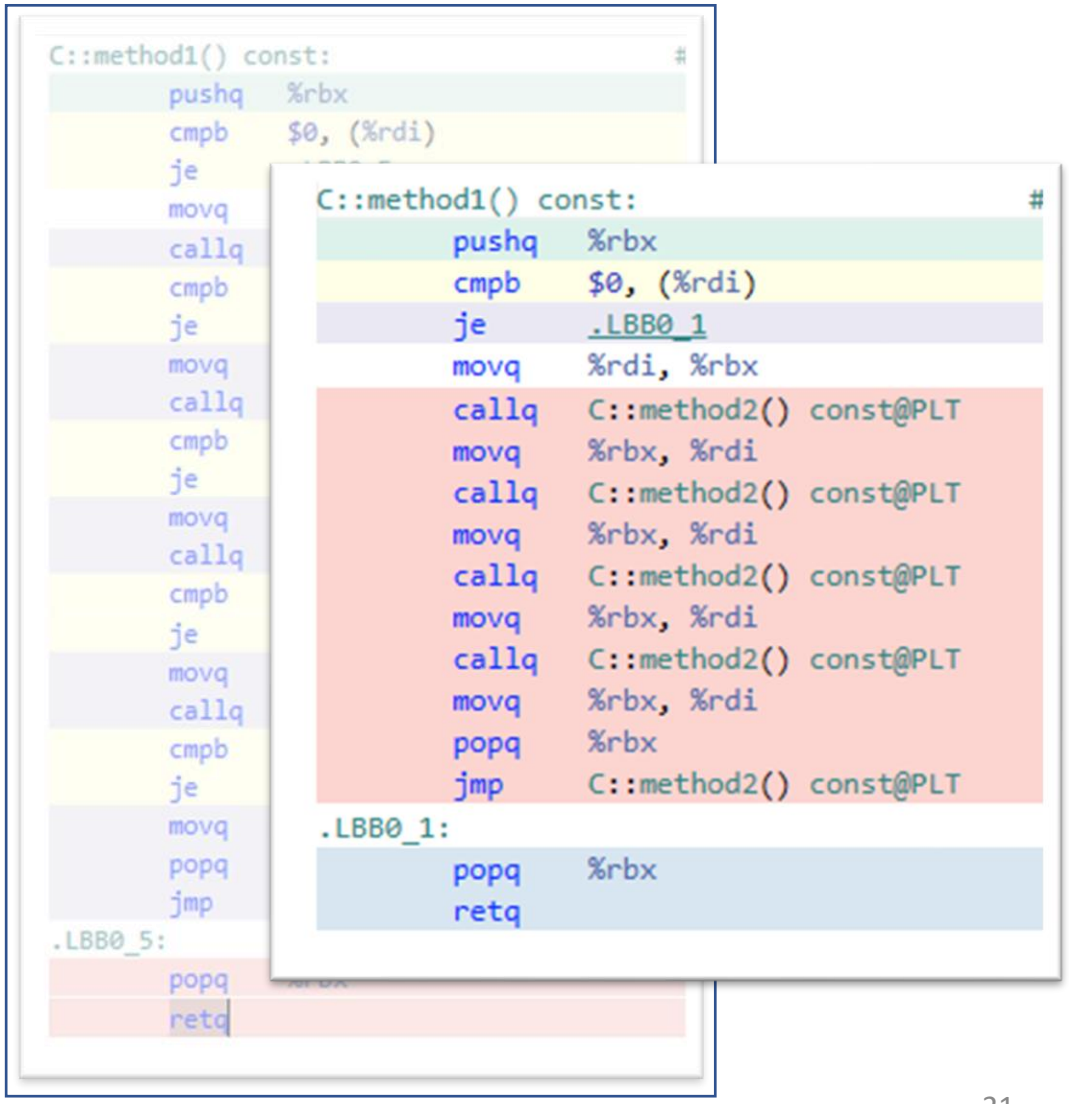


4. “Failed to move load with loop invariant address”

https://compiler-explorer.com/z/ve754bMe6

```
class C {
    const bool m_cond = true;
    void method1() const;
    void method2() const;
};

void C::method1() const {
    auto cond = m_cond;
    for(int i=0; i<5; ++i) {
        if (cond)
            method2();
    }
}
```



Part 2:
Getting Opt-Remarks
On Real Projects

-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

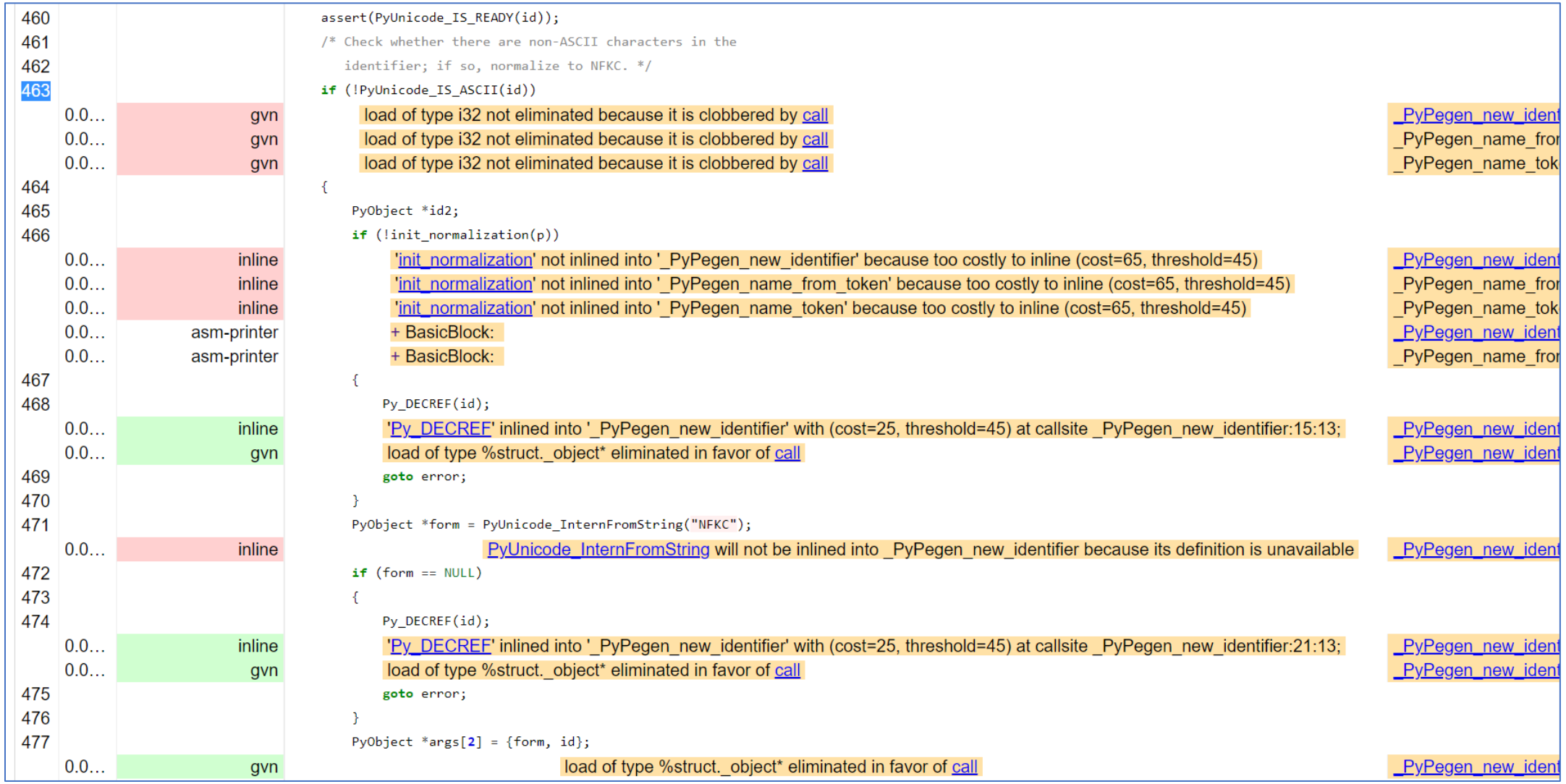
```
< /tmp/v.c
 2           | void bar();
 3           | void foo() { bar(); }
 4           |
 5           | void Test(int *res, int *c, int *d, int *p, int n) {
 6           |   int i;
 7           |
 8           | #pragma clang loop vectorize(assume_safety)
 9      V4,2 |   for (i = 0; i < 1600; i++) {
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

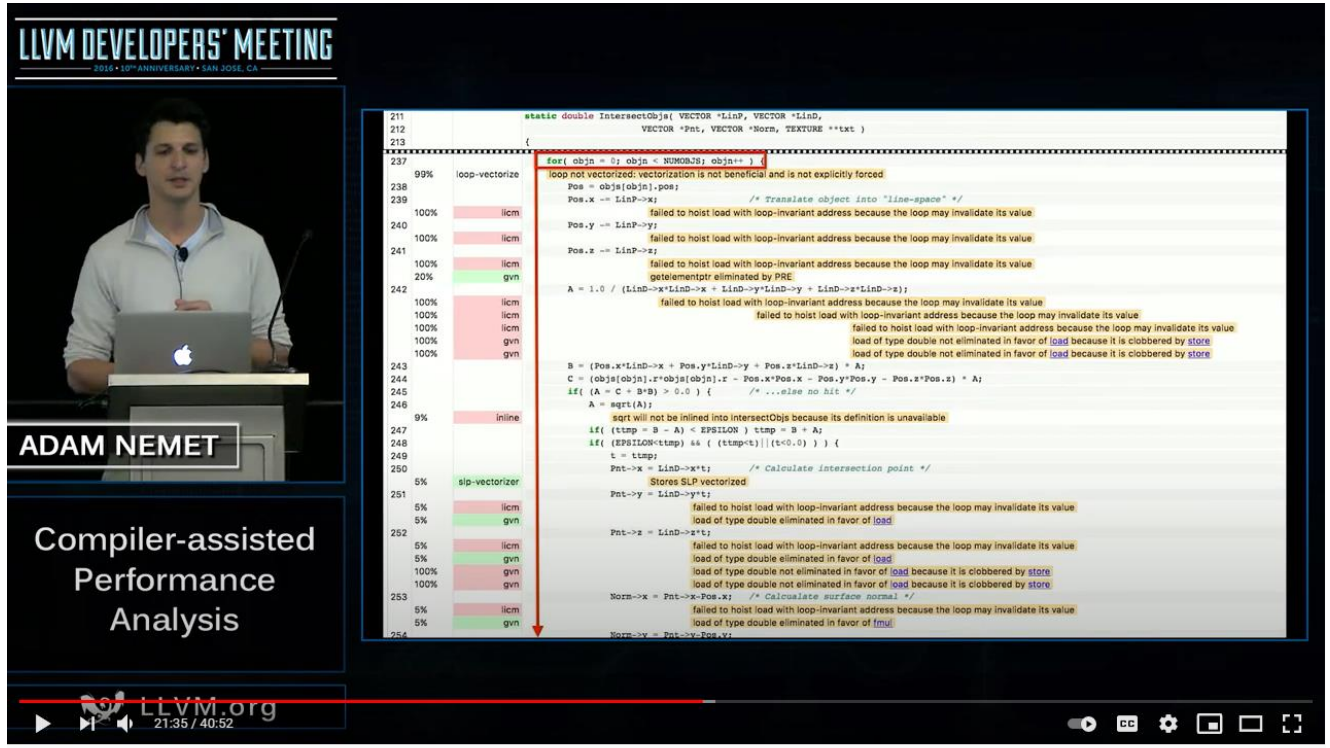
opt-viewer

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

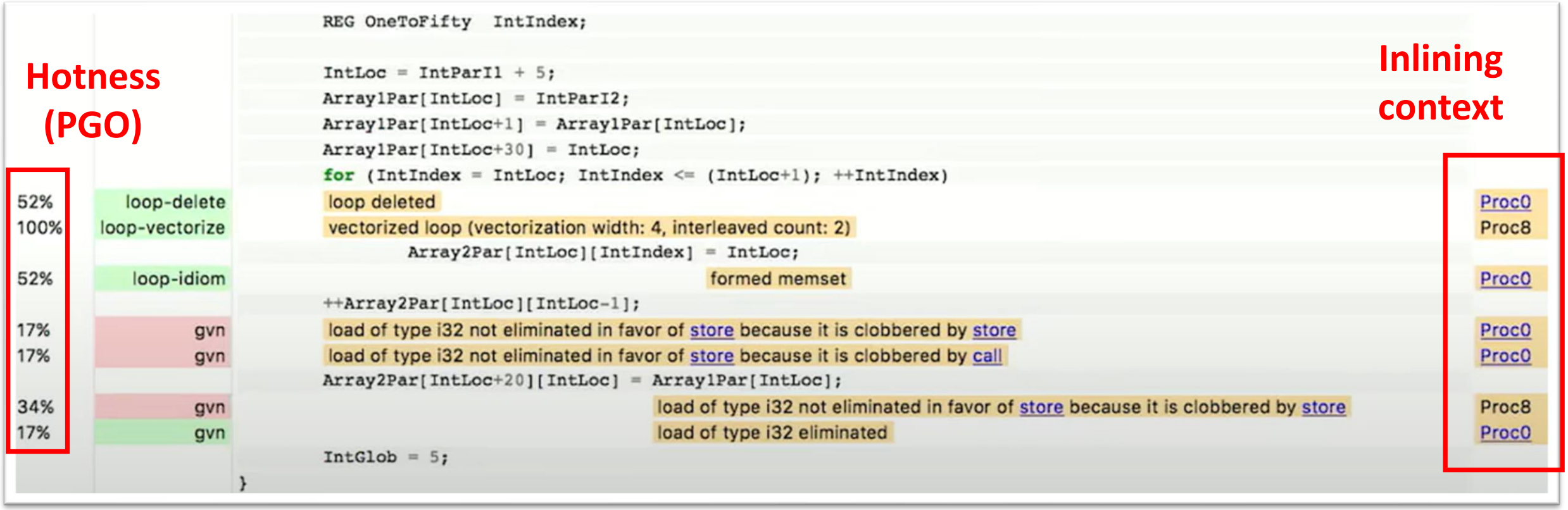
- Build with an extra clang switch:
  `-fsave-optimization-record`
  `*.opt.yaml` files are generated, by default in the obj folder.

- Generate htmls:

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

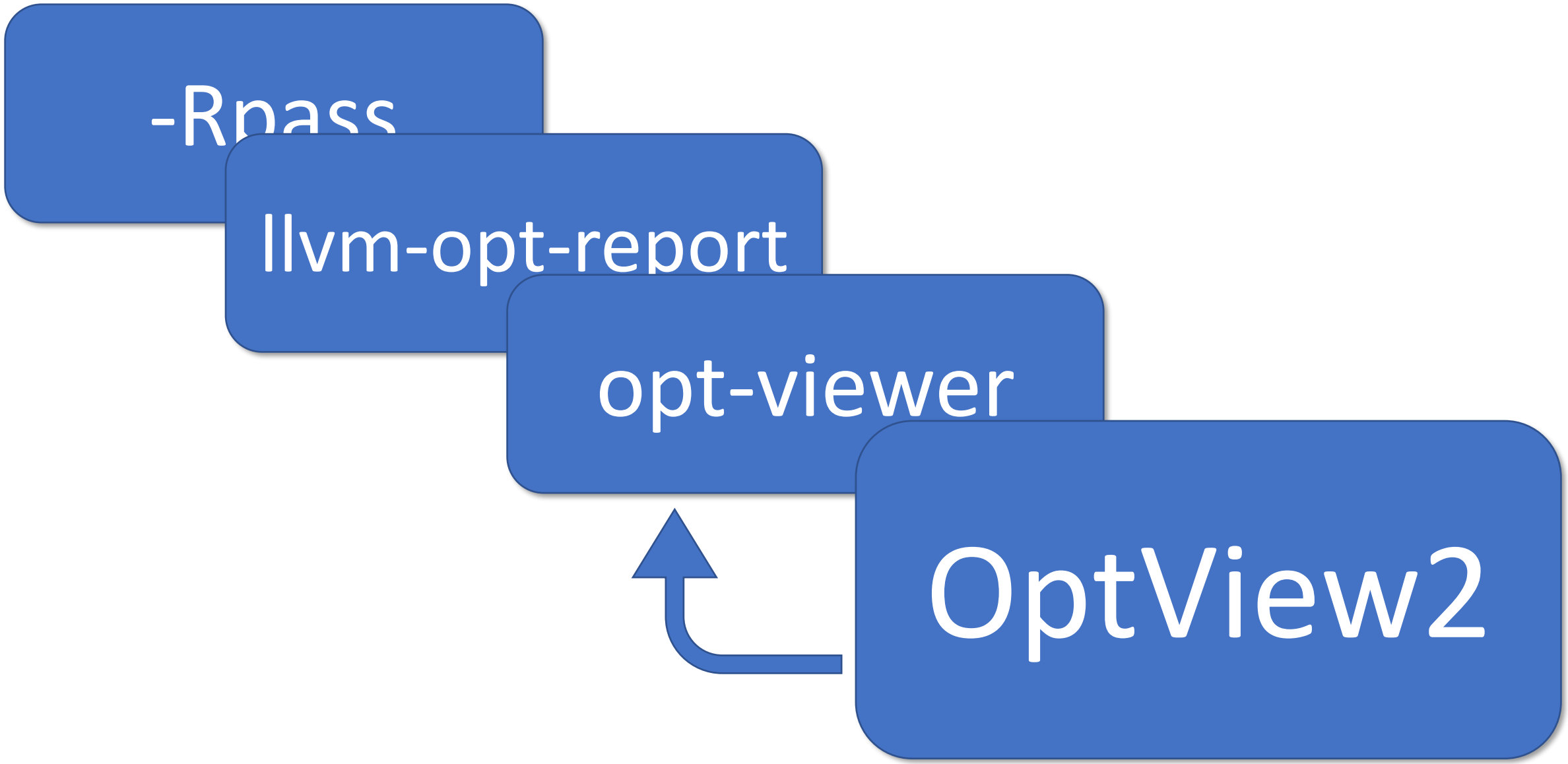
```
...
--- !Passed
Pass:            inline
Name:            Inlined
DebugLoc:        { File: '/usr/bin/../lib/gcc/
                   Line: 147, Column: 16 }
Function:        _ZNSt14pointer_traitsIPKcE10p
Args:
  - Callee:          _ZSt9addressofIKcEPT_RS1_
    DebugLoc:        { File: '/usr/bin/../lib/
                       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 output



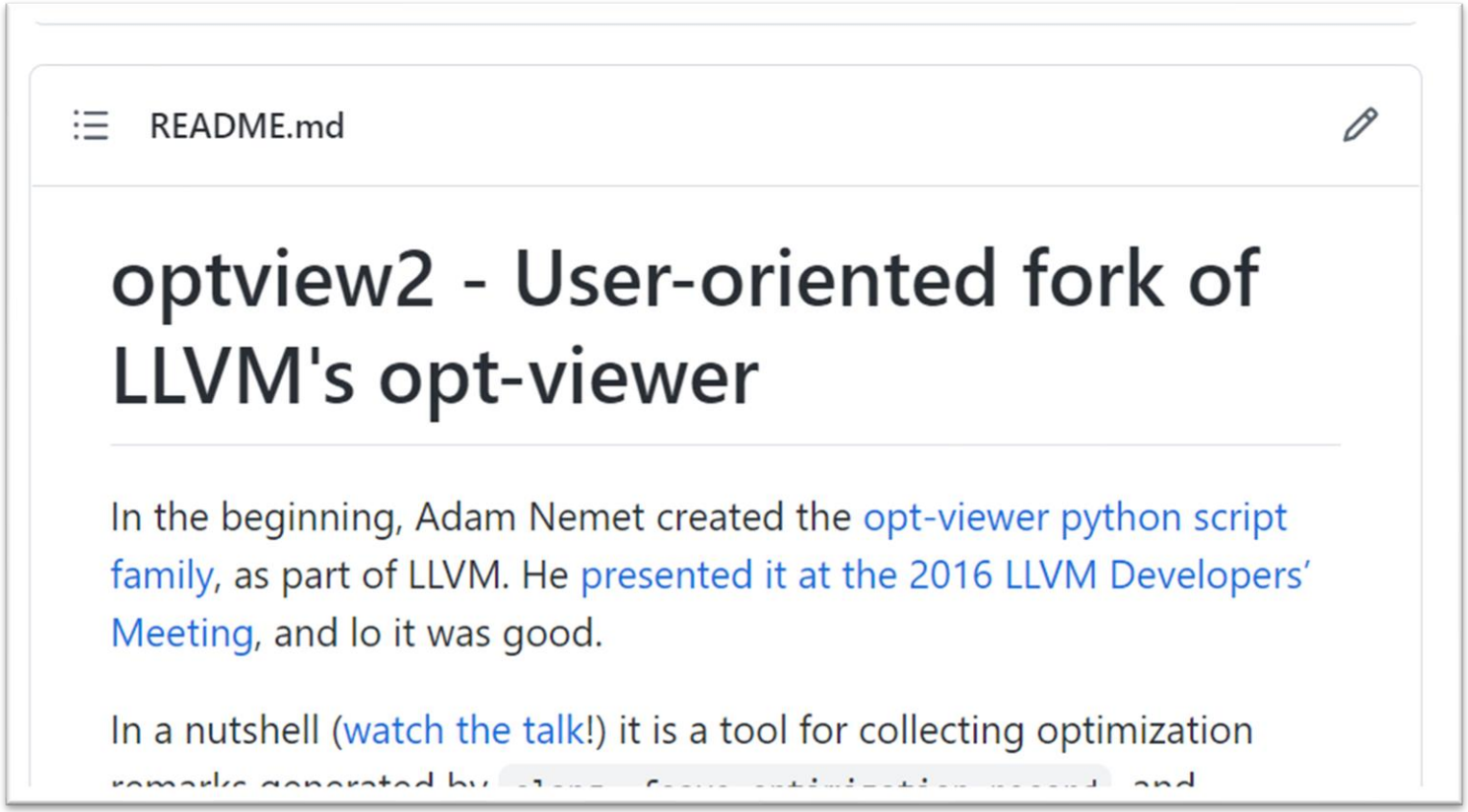
opt-viewer is great (seriously), but

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


-Rpass
llvm-opt-report
opt-viewer
OptView2

OptView2

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



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
- Sortable, resizable & pageable index
- Display column info (location within line)
- …

Part 3:
Beyond Classical
Clang Toolchain

GCC

- -fopt-info[-missed]

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

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

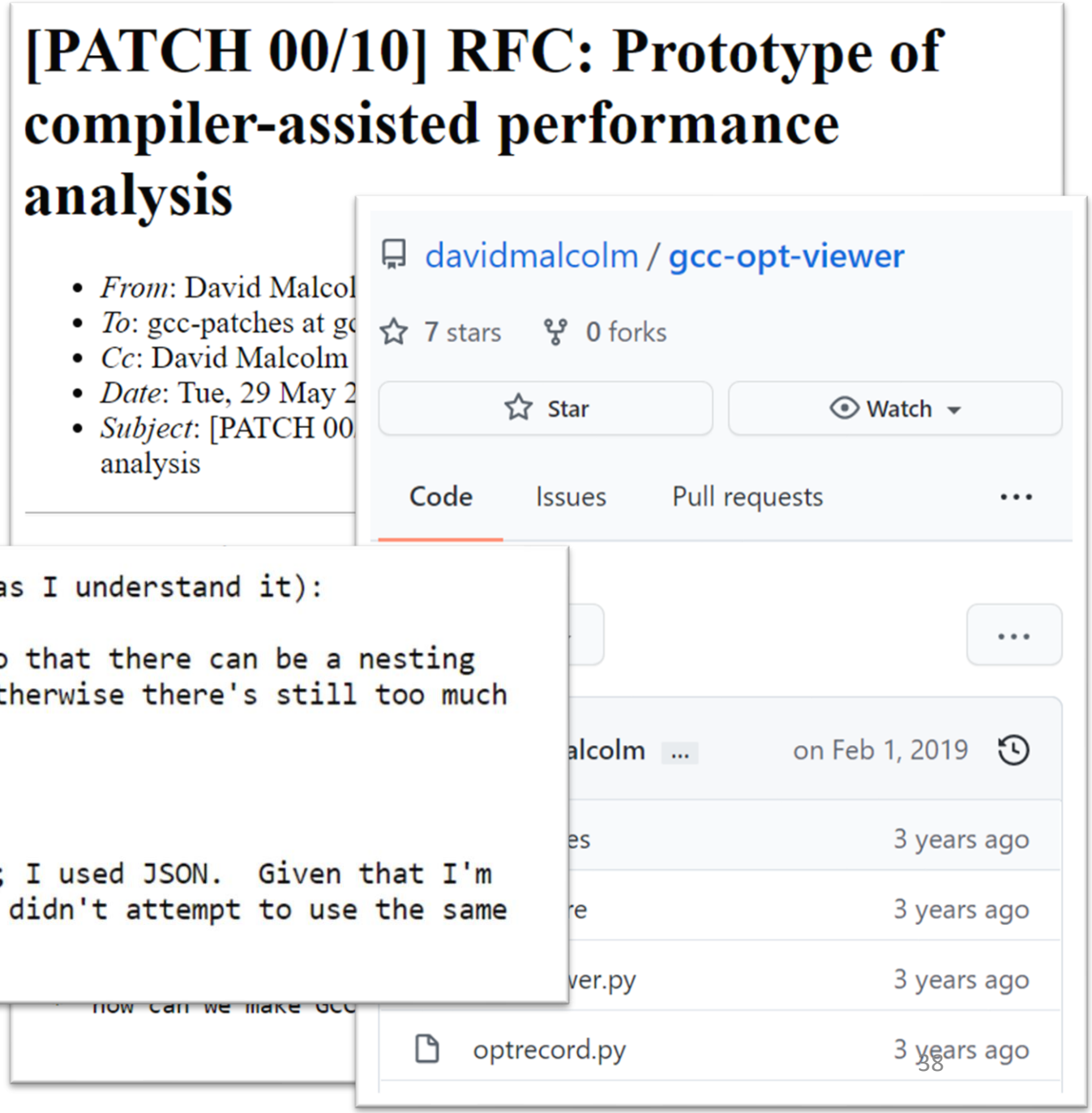
Output of x86-64 gcc 14.1 (Compiler #1)

A ▾ Wrap lines Select all

```
<source>:11:21: missed:   not inlinable: void f(int, int*)/0 -> int whateva()/2, function body not available
<source>:9:21: missed:   not inlinable: void f(int, int*)/0 -> int whateva()/2, function body not available
<source>:7:21: missed:   not inlinable: void f(int, int*)/0 -> int whateva()/2, function body not available
<source>:5:13: missed:   not inlinable: void f(int, int*)/0 -> void somefunc(const int&)/1, function body not available
<source>:5:13: missed: statement clobbers memory: somefunc (&i);
<source>:7:21: missed: statement clobbers memory: _11 = whateva ();
<source>:9:21: missed: statement clobbers memory: _16 = whateva ();
<source>:11:21: missed: statement clobbers memory: _20 = whateva ();
Compiler returned: 0
```

GCC

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



GCC

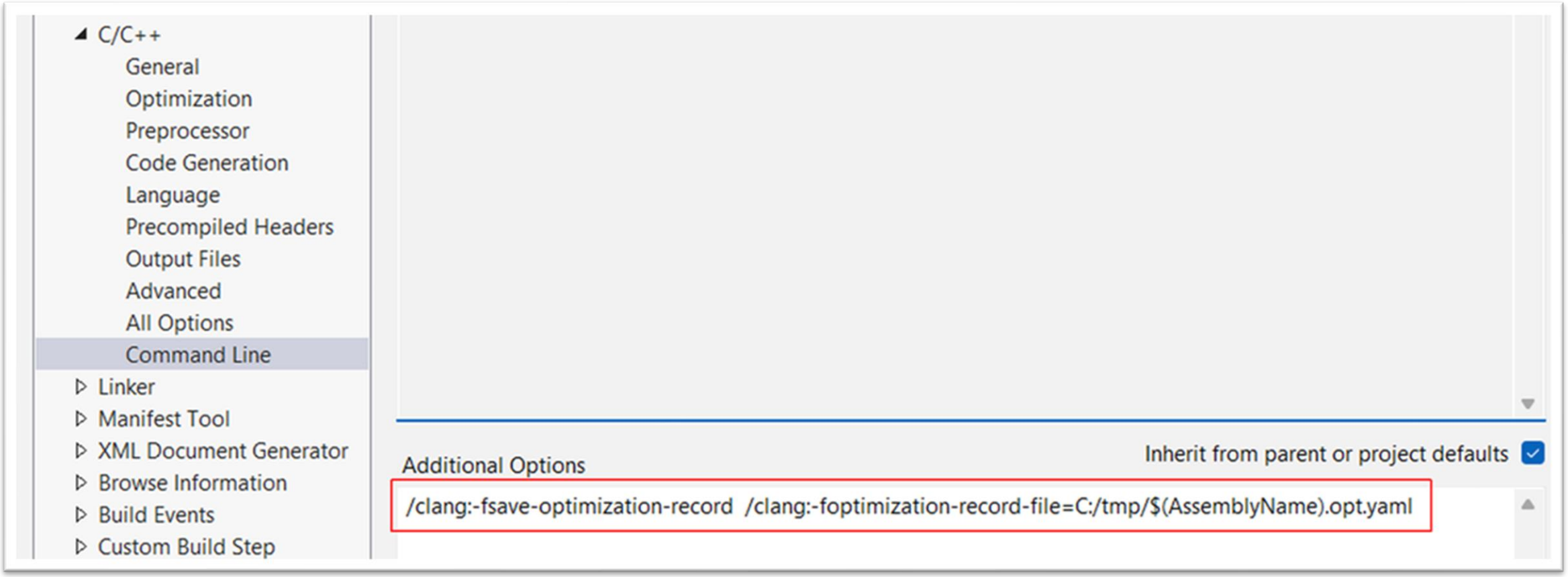
- https://dmalcolm.fedorapeople.org/gcc/2018-05-18/pgo-demo-test/pgo-demo-test/

6			{	
7			int sum = 0;	
8			for (int i = 0; i < n; ++i)	
	100.00	vect	^=== analyzing loop === === analyze_loop_nest === === vect_analyze_loop_form === === get_loop_niters === symbolic number of iterations is (unsigned int) n_9(D) not vectorized: loop contains function calls or data references that cannot be analyzed	compute_sum_without_inlining
	100.00	vect	^vectorized 0 loops in function	compute_sum_without_inlining
9			accumulate (arr[i], &sum);	
	100.00	inline	^not inlinable: compute_sum_without_inlining/0 -> accumulate/1, function body not available	compute_sum_without_inlining
10			return sum;	

GCC

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

Clang-cl usage



LTO

"inter-procedural analyses are often less precise … In LLVM, intra-procedural analyses are dominating in numbers and potential. "
Doerfert, Homerding, Finkel
ttps://github.com/jdoerfert/PETOSPA/blob/master/ISC19.pdf

"As for the fat LTO pipeline … well, we don't talk about the fat LTO pipeline. Its design is somewhere between non-sensical and non-existent." Nikita Popov
https://www.npopov.com/2023/04/07/LLVM-middle-end-pipeline.html

- Inlining – very visible improvement.
- Escape & Aliasing – no visible improvement.

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.**

Decorations across compilers

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

* Pertains also to locals

** Decorates a function return value

*** In gcc/clang `const` functions should never take pointer/reference arguments

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)
- Can be used to find analogues in other compilers (Sun pragmas etc.)

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

Impact?

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

Final Musing

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

- Sometimes.

Ofek Shilon

Compiler-Explorer Maintainer

Dev @Speedata

ofekshilon@gmail.com

OfekShilon

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

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

Const method that modifies members

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

- UB
- “load eliminated in favor of undef” (passed): https://compiler-explorer/z/j6bsbT39n