



# Optimization Remarks

Helping the Compiler Generate Better Code

OFEK SHILON



20  
22



September 12th-16th



# Ofek Shilon

Developer  
@Istra Research

[ofekshilon@gmail.com](mailto: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 [-Rpass-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     |     I   foo(); bar(); foo();
20     |     I   ^
21     |         ^
22     |     }
23     |
24     | }
```



-Rpass

llvm-opt-report

opt-viewer

# opt-viewer sample output

460			assert(PyUnicode_IS_READY(id));		
461			/* Check whether there are non-ASCII characters in the		
462			identifier; if so, normalize to NFKC. */		
463	0.0...	gvn	if (!PyUnicode_IS_ASCII(id))	load of type i32 not eliminated because it is clobbered by <a href="#">call</a>	<a href="#">PyPegen_new_ident</a>
	0.0...	gvn		load of type i32 not eliminated because it is clobbered by <a href="#">call</a>	<a href="#">_PyPegen_name_from</a>
	0.0...	gvn		load of type i32 not eliminated because it is clobbered by <a href="#">call</a>	<a href="#">_PyPegen_name_tok</a>
464			{		
465			PyObject *id2;		
466	0.0...	inline	if (!init_normalization(p))	'init_normalization' not inlined into '_PyPegen_new_identifier' because too costly to inline (cost=65, threshold=45)	<a href="#">PyPegen_new_ident</a>
	0.0...	inline		'init_normalization' not inlined into '_PyPegen_name_from_token' because too costly to inline (cost=65, threshold=45)	<a href="#">_PyPegen_name_from</a>
	0.0...	inline		'init_normalization' not inlined into '_PyPegen_name_token' because too costly to inline (cost=65, threshold=45)	<a href="#">_PyPegen_name_tok</a>
	0.0...	asm-printer	+ BasicBlock:		<a href="#">PyPegen_new_ident</a>
	0.0...	asm-printer	+ BasicBlock:		<a href="#">_PyPegen_name_from</a>
467			{		
468	0.0...	inline	Py_DECREF(id);	'Py_DECREF' inlined into '_PyPegen_new_identifier' with (cost=25, threshold=45) at callsite _PyPegen_new_identifier:15:13;	<a href="#">PyPegen_new_ident</a>
	0.0...	gvn		load of type %struct._object* eliminated in favor of <a href="#">call</a>	<a href="#">_PyPegen_new_ident</a>
469			goto error;		
470			}		
471	0.0...	inline	PyObject *form = PyUnicode_InternFromString("NFKC");	<a href="#">PyUnicode_InternFromString</a> will not be inlined into _PyPegen_new_identifier because its definition is unavailable	<a href="#">PyPegen_new_ident</a>
472			if (form == NULL)		
473			{		
474	0.0...	inline	Py_DECREF(id);	'Py_DECREF' inlined into '_PyPegen_new_identifier' with (cost=25, threshold=45) at callsite _PyPegen_new_identifier:21:13;	<a href="#">PyPegen_new_ident</a>
	0.0...	gvn		load of type %struct._object* eliminated in favor of <a href="#">call</a>	<a href="#">_PyPegen_new_ident</a>
475			goto error;		
476			}		
477	0.0...	gvn	PyObject *args[2] = {form, id};	load of type %struct._object* eliminated in favor of <a href="#">call</a>	<a href="#">_PyPegen_new_ident</a>

# 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

The screenshot shows a video player with a dark theme. On the left, a small video window shows Adam Nemet speaking at a podium. Below this, the text 'ADAM NEMET' and 'Compiler-assisted Performance Analysis' is displayed. The main part of the screen shows a code editor with LLVM IR for a function named 'IntersectObj'. The code is annotated with various optimization flags and their results, such as 'loop-vectorize', 'licm', 'gvn', and 'slp-vectorizer'. A red vertical line highlights a specific section of the code. The video player controls at the bottom show the video is at 21:35 / 40:52. The title '2016 LLVM Developers' Meeting: A. Nemet "Compiler-assisted Performance Analysis"' and view count '1,824 views • Dec 2, 2016' are visible.

LLVM DEVELOPERS' MEETING  
2016 • 10<sup>th</sup> ANNIVERSARY • SAN JOSE, CA

ADAM NEMET

Compiler-assisted  
Performance  
Analysis

LLVM.org  
21:35 / 40:52

2016 LLVM Developers' Meeting: A. Nemet "Compiler-assisted Performance Analysis"

1,824 views • Dec 2, 2016

25 0 SHARE SAVE ...

# 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

Hotness  
(PGO)

Inlining  
context

52%	loop-delete	REG OneToFifty IntIndex;	
100%	loop-vectorize	IntLoc = IntParI1 + 5;	
		Array1Par[IntLoc] = IntParI2;	
		Array1Par[IntLoc+1] = Array1Par[IntLoc];	
		Array1Par[IntLoc+30] = IntLoc;	
		for (IntIndex = IntLoc; IntIndex <= (IntLoc+1); ++IntIndex)	
		loop deleted	
		vectorized loop (vectorization width: 4, interleaved count: 2)	
		Array2Par[IntLoc][IntIndex] = IntLoc;	
		formed memset	
		++Array2Par[IntLoc][IntLoc-1];	
17%	gvn	load of type i32 not eliminated in favor of store because it is clobbered by store	Proc0
17%	gvn	load of type i32 not eliminated in favor of store because it is clobbered by call	Proc8
		Array2Par[IntLoc+20][IntLoc] = Array1Par[IntLoc];	
34%	gvn	load of type i32 not eliminated in favor of store because it is clobbered by store	Proc0
17%	gvn	load of type i32 eliminated	Proc0
		IntGlob = 5;	
		}	

-Rpass

llvm-opt-report

opt-viewer

OptView2



```
graph LR; OptView2 --> opt-viewer
```

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

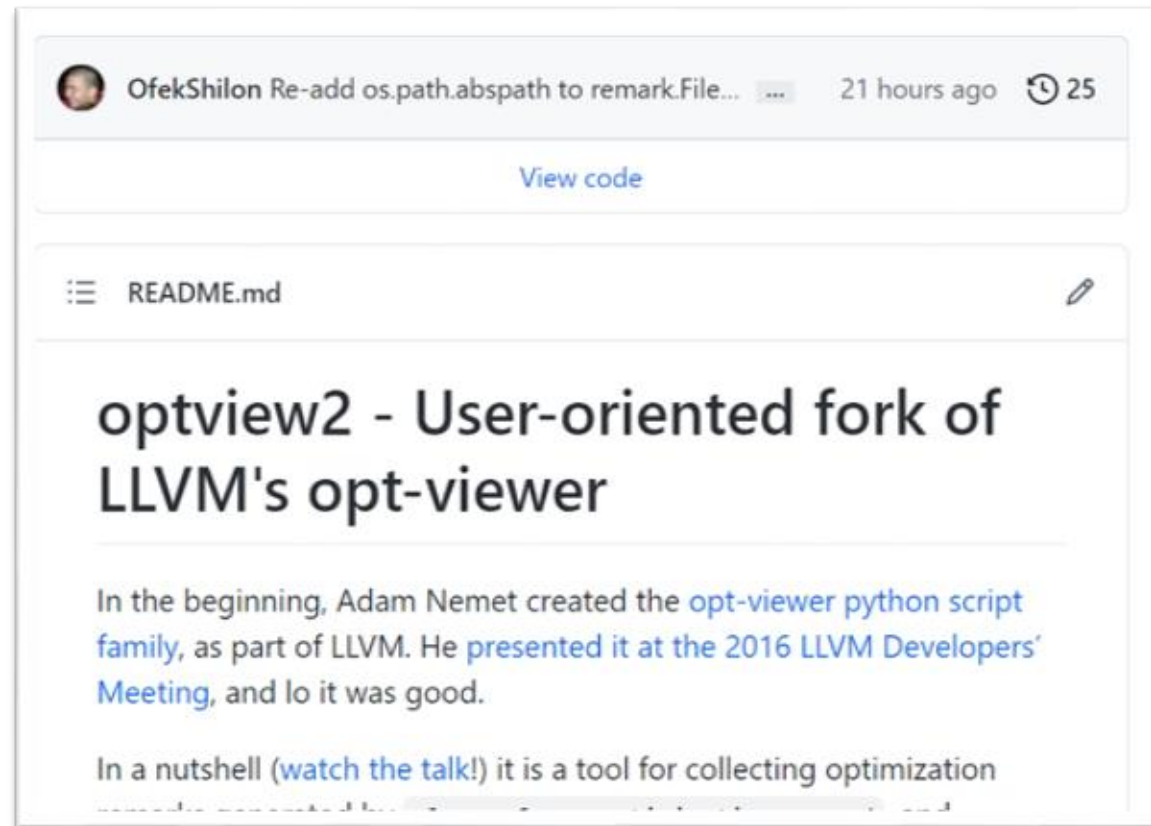


# Improvements in OptView2

- 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!

The screenshot shows the Compiler Explorer interface. The top bar displays 'x86-64 clang 14.0.0' with a red box around 'clang'. Below it, the 'Add new...' button is also highlighted with a red box. A dropdown menu is open, showing various compiler views, with 'Optimization' (represented by a clock icon) highlighted and a red arrow pointing to it. The main editor shows assembly code for a function 'foo'. To the right, a smaller window titled 'Opt Viewer x86-64 clang 12.0.1' displays the C++ source code for the same function. Below this window, two callout boxes provide analysis results:

- Missed** - failed to move load with loop-invariant address because the loop may invalidate its value
- Missed** - load of type i32 not eliminated in favor of load (3:17) because it is clobbered by store (3:14)

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

```
• Load of type double not eliminated because it is clobbered by call  
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 costly to inline (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 }
```

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)

```
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)  
16    movl    (%rsi), %eax  
17    addl    %eax, 28(%rdi)  
18    movl    (%rsi), %eax  
19    addl    %eax, 32(%rdi)
```



## 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)

A

```
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     paddb  %xmm0, %xmm1  
8     movdqu  %xmm1, (%rdi)  
9     paddb  %xmm0, %xmm2  
10    movdqu  %xmm2, 16(%rdi)  
11    addl    %eax, 32(%rdi)  
12    addl    %eax, 36(%rdi)  
13    retq
```

```
15    addl    %eax, 24(%rdi)  
16    movl    (%rsi), %eax  
17    addl    %eax, 28(%rdi)  
18    movl    (%rsi), %eax  
19    addl    %eax, 32(%rdi)
```



## 2. "Clobbered by store"

The image shows a C++ IDE with three overlapping windows. The top window displays the source code of a function `foo` with parameters `int* a` and `const int& b`. The middle window shows a modified version of the function where the parameter `a` is `long*`, with `long*` highlighted by a red box. The bottom window shows the assembly output for the modified function, also with `long*` highlighted by a red box. The assembly code shows instructions for moving pointers and performing arithmetic on memory.

```
1 void foo(int* a, const int& b) {  
2  
3  
4  
5 }  
6
```

```
1 void foo(long* a, const int& b) {  
2     for (int i=0; i<16; i++) {  
3         a[i] += b;  
4     }  
5 }  
6
```

```
1 void foo(long* a, const int& b) {  
2     for (int i=0; i<16; i++) {  
3         a[i] += b;  
4     }  
5 }  
6
```

Assembly output for `foo(long*, int const&):`

```
1  
2     mov     eax, dword ptr [esp + 4]  
3     mov     ecx, dword ptr [esp + 8]  
4     movd    xmm0, dword ptr [ecx]      #  
5     pshufd  xmm0, xmm0, 0             #  
6     movdqu  xmm1, xmmword ptr [eax]  
7     movdqu  xmm2, xmmword ptr [eax + 16]  
8     padd    xmm1, xmm0  
9     movdqu  xmmword ptr [eax], xmm1  
10    padd    xmm2, xmm0  
11    movdqu  xmmword ptr [eax + 16], xmm2  
12    movdqu  xmm1, xmmword ptr [eax + 32]  
13    padd    xmm1, xmm0  
14    movdqu  xmmword ptr [eax + 32], xmm1  
15    movdqu  xmm1, xmmword ptr [eax + 48]  
16    padd    xmm1, xmm0  
17    movdqu  xmmword ptr [eax + 48], xmm1  
18    ret
```

## 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 whatever();
```

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

```
1 void somefunc(const int&);  
2 int whatever();  
3  
4 void f(int i, int* res) {  
5     somefunc(i);  
6     i++;  
7     res[0] = whatever();  
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)  
9     callq    whatever()@PLT  
10    movl     %eax, (%rbx)  
11    incl     12(%rsp)  
12    callq    whatever()@PLT  
13    movl     %eax, 4(%rbx)  
14    incl     12(%rsp)  
15    callq    whatever()@PLT  
16    movl     %eax, 8(%rbx)  
17    addq     $16, %rsp  
18    retq
```

**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 whatever();

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

```
1  f(int, int*):
2      pushq    %rbx
3      subq    $16, %rsp
4
5  1  f(int, int*):
6      pushq    %rbx
7      movq     %rsi, %rbx
8      callq    whatever()@PLT
9      movl     %eax, (%rbx)
10     callq    whatever()@PLT
11     movl     %eax, 4(%rbx)
12     callq    whatever()@PLT
13     movl     %eax, 8(%rbx)
14     popq     %rbx
15     retq
16
17     addq     $16, %rsp
18     popq     %rbx
19     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 whatever() __attribute__((const));  
  
void f(int i, int* res) {  
    somefunc(i);  
    i++;  
    res[0] = whatever();  
    i++;  
    res[1] = whatever();  
    i++;  
    res[2] = whatever();  
}
```

```
1  f(int, int*):  
2      pushq    %rbx  
3  
4  1  f(int, int*):  
5  2      pushq    %rbx  
6  3      subq     $16, %rsp  
7  4      movq     %rsi, %rbx  
8  5      movl     %edi, 12(%rsp)  
9  6      leaq     12(%rsp), %rdi  
10 7      callq    somefunc(int const&)  
11 8      callq    whatever()@PLT  
12 9      movl     %eax, (%rbx)  
13 10     movl     %eax, 4(%rbx)  
14 11     movl     %eax, 8(%rbx)  
15 12     addq     $16, %rsp  
16 13     popq     %rbx  
17 14     retq
```

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

### 3. “Clobbered by call”

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

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

```
1  f(int, int*):  
2  
3  
4
```

```
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    whatever()@PLT  
9      movl     %eax, (%rbx)  
10     callq    whatever()@PLT  
11     movl     %eax, 4(%rbx)  
12     callq    whatever()@PLT  
13     movl     %eax, 8(%rbx)  
14     addq     $16, %rsp  
15     popq     %rbx  
16     retq
```

```
18  
19
```

### 3. “Clobbered by call”

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

```
1  f(int, int*):  
2      pushq    %rbx  
3  
4      1  f(int, int*):  
5      2      pushq    %rbx  
6      3      subq    $16, %rsp  
7      4      movq    %rsi, %rbx  
8      5      movl    %edi, 12(%rsp)  
9      6      leaq    12(%rsp), %rdi  
10     7      callq   somefunc(int const&  
11     8      callq   whatever()@PLT  
12     9      movl    %eax, (%rbx)  
13    10      callq   whatever()@PLT  
14    11      movl    %eax, 4(%rbx)  
15    12      callq   whatever()@PLT  
16    13      movl    %eax, 8(%rbx)  
17    14      addq    $16, %rsp  
18    15      popq    %rbx  
19    16      retq
```



### 3. “Clobbered by call”

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

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

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

Opt Viewer x86-64 clang 13.0.0 (Editor #1, Compiler #1) ✎ ✕  
A ▾  
4 void f(int i) {  
5 std::ofstream fs("myfile");  
6 fs << &i;  
7 i++;  
8 whatever();  
9 i++;  
10 whatever();  
11 whatever();  
12 whatever();  
13 }

Missed - load of type i32 not eliminated in favor of store (9:6)  
because it is clobbered by invoke (10:5)

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

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

<https://godbolt.org/z/YGc83TMnj>

```
class C {  
    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();  
    }  
}
```

Address	Disassembly	Comment
00401000	push rbp	
00401001	cmp byte ptr [rbp+4], 0	
00401002	je .LBB0_1	
00401003	mov rbp, rdi	
00401004	call f(0)	
00401005	cmp byte ptr [rbp+4], 0	
00401006	je .LBB0_1	
00401007	call f(0)	
00401008	cmp byte ptr [rbp+4], 0	
00401009	je .LBB0_1	
0040100A	call f(0)	
0040100B	cmp byte ptr [rbp+4], 0	
0040100C	je .LBB0_1	
0040100D	call f(0)	
0040100E	cmp byte ptr [rbp+4], 0	
0040100F	je .LBB0_1	
00401010	call f(0)	
00401011	cmp byte ptr [rbp+4], 0	
00401012	je .LBB0_1	
00401013	call f(0)	
00401014	cmp byte ptr [rbp+4], 0	
00401015	je .LBB0_1	
00401016	call f(0)	
00401017	cmp byte ptr [rbp+4], 0	
00401018	je .LBB0_1	
00401019	call f(0)	
0040101A	cmp byte ptr [rbp+4], 0	
0040101B	je .LBB0_1	
0040101C	call f(0)	
0040101D	cmp byte ptr [rbp+4], 0	
0040101E	je .LBB0_1	
0040101F	call f(0)	
00401020	cmp byte ptr [rbp+4], 0	
00401021	je .LBB0_1	
00401022	call f(0)	
00401023	cmp byte ptr [rbp+4], 0	
00401024	je .LBB0_1	
00401025	call f(0)	
00401026	cmp byte ptr [rbp+4], 0	
00401027	je .LBB0_1	
00401028	call f(0)	
00401029	cmp byte ptr [rbp+4], 0	
0040102A	je .LBB0_1	
0040102B	call f(0)	
0040102C	cmp byte ptr [rbp+4], 0	
0040102D	je .LBB0_1	
0040102E	call f(0)	
0040102F	cmp byte ptr [rbp+4], 0	
00401030	je .LBB0_1	
00401031	call f(0)	
00401032	cmp byte ptr [rbp+4], 0	
00401033	je .LBB0_1	
00401034	call f(0)	
00401035	cmp byte ptr [rbp+4], 0	
00401036	je .LBB0_1	
00401037	call f(0)	
00401038	cmp byte ptr [rbp+4], 0	
00401039	je .LBB0_1	
0040103A	call f(0)	
0040103B	cmp byte ptr [rbp+4], 0	
0040103C	je .LBB0_1	
0040103D	call f(0)	
0040103E	cmp byte ptr [rbp+4], 0	
0040103F	je .LBB0_1	
00401040	call f(0)	
00401041	cmp byte ptr [rbp+4], 0	
00401042	je .LBB0_1	
00401043	call f(0)	
00401044	cmp byte ptr [rbp+4], 0	
00401045	je .LBB0_1	
00401046	call f(0)	
00401047	cmp byte ptr [rbp+4], 0	
00401048	je .LBB0_1	
00401049	call f(0)	
0040104A	cmp byte ptr [rbp+4], 0	
0040104B	je .LBB0_1	
0040104C	call f(0)	
0040104D	cmp byte ptr [rbp+4], 0	
0040104E	je .LBB0_1	
0040104F	call f(0)	
00401050	cmp byte ptr [rbp+4], 0	
00401051	je .LBB0_1	
00401052	call f(0)	
00401053	cmp byte ptr [rbp+4], 0	
00401054	je .LBB0_1	
00401055	call f(0)	
00401056	cmp byte ptr [rbp+4], 0	
00401057	je .LBB0_1	
00401058	call f(0)	
00401059	cmp byte ptr [rbp+4], 0	
0040105A	je .LBB0_1	
0040105B	call f(0)	
0040105C	cmp byte ptr [rbp+4], 0	
0040105D	je .LBB0_1	
0040105E	call f(0)	
0040105F	cmp byte ptr [rbp+4], 0	
00401060	je .LBB0_1	
00401061	call f(0)	
00401062	cmp byte ptr [rbp+4], 0	
00401063	je .LBB0_1	
00401064	call f(0)	
00401065	cmp byte ptr [rbp+4], 0	
00401066	je .LBB0_1	

```
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
```

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

```

    pop     rbx
    ret

```

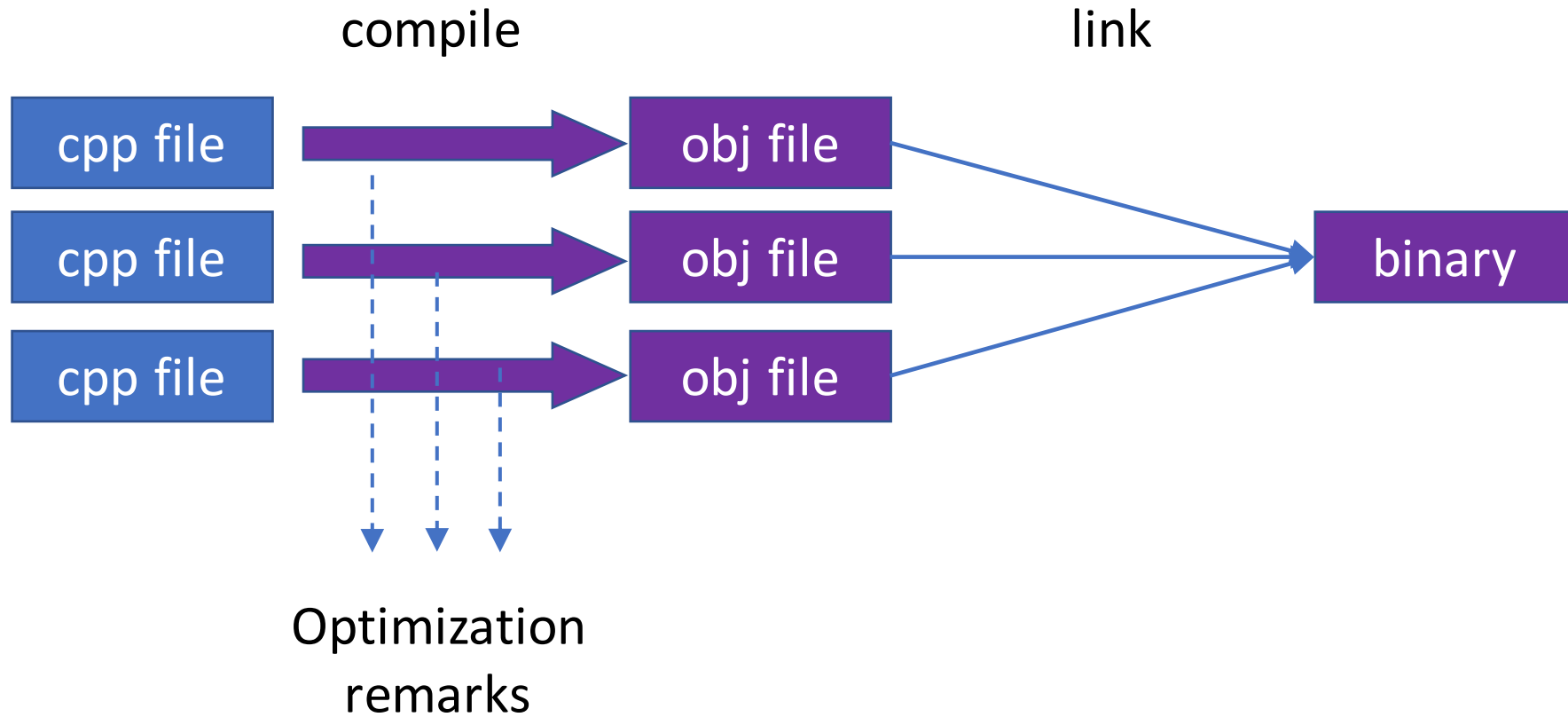
# Cheat Sheet

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 <i>before</i> 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

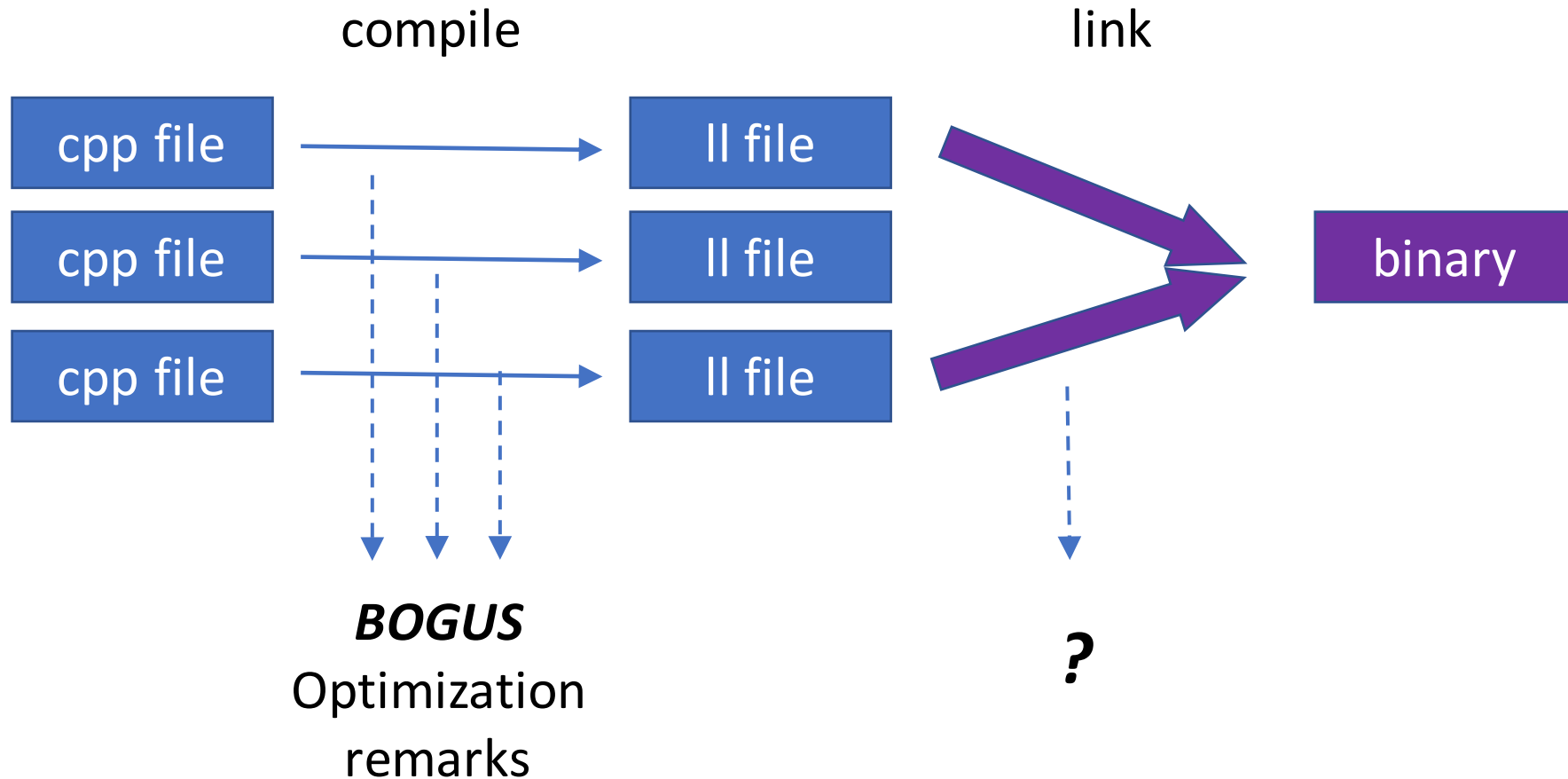
# Opt remarks with LTO

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>

## [PATCH 00/10] RFC: Prototype of compiler-assisted performance analysis

- *From:* David Malcolm
- *To:* gcc-patches at gcc.gnu.org
- *Cc:* David Malcolm
- *Date:* Tue, 29 May 2018 15:00:00 -0400
- *Subject:* [PATCH 00/10] RFC: Prototype of compiler-assisted performance analysis

📄 davidmalcolm / gcc-opt-viewer

☆ 7 stars 🍴 0 forks

☆ Star

👁 Watch ▾

Code

Issues

Pull requests

...

Differences from Adam Nemet's work (as I understand it):

- \* I've added hierarchical records, so that there can be a nesting structure of optimization notes (otherwise there's still too much of a "wall of text").
- \* capture of GCC source location
- \* LLVM is using YAML for some reason; I used JSON. Given that I'm capturing some different things, I didn't attempt to use the same file format as LLVM.

now can we make gcc

📄 optrecord.py

3 years ago

# GCC work

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

5			{	
7			int sum = 0;	
3			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_with out_inlining
	100.00	vect	^vectorized 0 loops in function	compute_sum_with out_inlining
9			accumulate (arr[i], &sum);	
	100.00	inline	^not inlinable: compute_sum_without_inlining/0 -> accumulate/1, function body not available	compute_sum_with out_inlining
10			return sum;	

# 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
<code>__restrict</code>	V	V	<code>__restrict *</code> <code>__declspec(restrict) **</code>
<code>__attribute__((pure))</code>	V	-	-
<code>__attribute__((const))</code>	V	V	<code>__declspec(noalias)</code>
<code>__attribute__((noescape))</code>	-	-	-

\* 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>

The image displays a Rust IDE with three panels. The left panel shows the Rust source code for a function `foo` and a boxed-in Rust function `foo2`. The middle panel shows the assembly code for `foo`. The right panel shows the LLVM IR for `example::foo2`.

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

pub fn foo2(a: &mut [i32; 10] , b: &i32) {
    for i in 0..10 {
        a[i] += *b;
    }
}
```

```
1 foo(int*, int const&):
2     movl    (%rsi), %eax
3     addl    $1, %eax
4     movl    %eax, %edi
5     addl    $1, %edi
6     movl    %edi, %eax
7     addl    $1, %eax
8     movl    %eax, %edi
9     addl    $1, %eax
10    movl    %eax, %edi
11    addl    $1, %eax
12    movl    %eax, %edi
13    addl    $1, %eax
14    movl    %eax, %edi
15    addl    $1, %eax
16    movl    %eax, %edi
17    addl    $1, %eax
18    movl    %eax, %edi
19    addl    $1, %eax
20    movl    %eax, %edi
```

```
1 example::foo2:
2     mov     eax, dword ptr [rsi]
3     movd    xmm0, eax
4     pshufd  xmm0, xmm0, 0
5     movdqu  xmm1, xmmword ptr [rdi]
6     movdqu  xmm2, xmmword ptr [rdi + 16]
7     paddq   xmm1, xmm0
8     movdqu  xmmword ptr [rdi], xmm1
9     paddq   xmm2, xmm0
10    movdqu  xmmword ptr [rdi + 16], xmm2
11    add     dword ptr [rdi + 32], eax
12    add     dword ptr [rdi + 36], eax
13    ret
```



# Carbon

Carries some good news about escape analysis:

<https://www.foonathan.net/2022/07/carbon-calling-convention/>

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

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

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

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

<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  $\mu$ s -> 4.6  $\mu$ s

# 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





# Optimization Remarks

Helping the Compiler Generate Better Code

OFEK SHILON



20  
22







## 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 // Assignment optimized properly:

```

```

5 void f1(S& s1, S& s2 ) {
6     s1 = s2;
7 }

```

```

8 // Assignment not optimized due to bogus potential
9 // aliasing between a and b (see opt remarks):

```

```

10 void f2(S& s1, S& s2) {
11     s1.a = s2.a;
12     s1.b = s2.b;
13 }

```

```

14 // Assignment optimized properly:

```

```

15 void f3(S& s1, S& s2) {
16     s1.a.t = s2.a.t;
17     s1.b.t = s2.b.t;
18 }

```

```

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

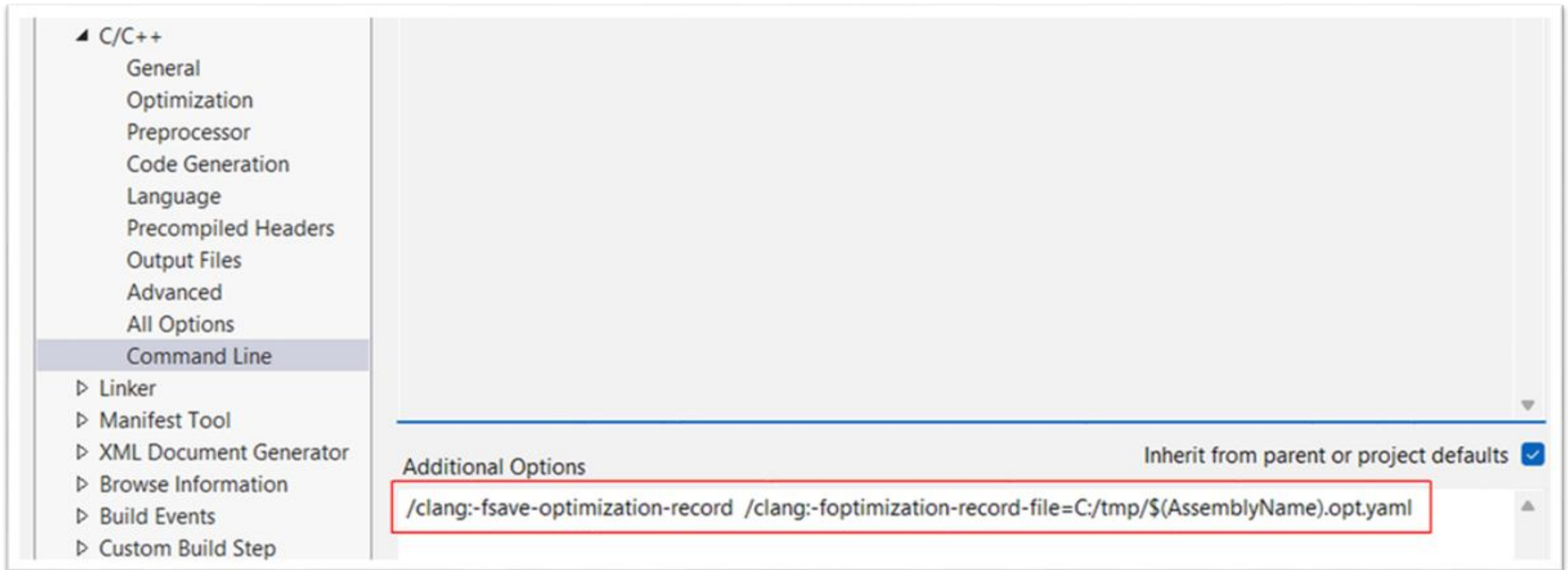
```

```

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  
};
```