New IPA-modref pass for GCC

Jan Hubička

Joint project with David Čepelík. Partly funded by AMD.

Sep 14, 2021
Inter-procedural optimization passes in GCC

1. Symbol visibility optimizations
Inter-procedural optimization passes in GCC

1. Symbol visibility optimizations
2. Unreachable function and variable removal
Inter-procedural optimization passes in GCC

1. Symbol visibility optimizations
2. Unreachable function and variable removal
3. Inter-procedural profile propagation and indirect call transformation
Inter-procedural optimization passes in GCC

1. Symbol visibility optimizations
2. Unreachable function and variable removal
3. Inter-procedural profile propagation and indirect call transformation
4. Identical code folding
Inter-procedural optimization passes in GCC

1. Symbol visibility optimizations
2. Unreachable function and variable removal
3. Inter-procedural profile propagation and indirect call transformation
4. Identical code folding
5. (Speculative) devirtualization
Inter-procedural optimization passes in GCC

1. Symbol visibility optimizations
2. Unreachable function and variable removal
3. Inter-procedural profile propagation and indirect call transformation
4. Identical code folding
5. (Speculative) devirtualization
6. Constant propagation and function specialization (cloning)
Inter-procedural optimization passes in GCC

1. Symbol visibility optimizations
2. Unreachable function and variable removal
3. Inter-procedural profile propagation and indirect call transformation
4. Identical code folding
5. (Speculative) devirtualization
6. Constant propagation and function specialization (cloning)
7. Constructor/destructor merging
Inter-procedural optimization passes in GCC

1. Symbol visibility optimizations
2. Unreachable function and variable removal
3. Inter-procedural profile propagation and indirect call transformation
4. Identical code folding
5. (Speculative) devirtualization
6. Constant propagation and function specialization (cloning)
7. Constructor/destructor merging
8. Inlining
Inter-procedural optimization passes in GCC

1. Symbol visibility optimizations
2. Unreachable function and variable removal
3. Inter-procedural profile propagation and indirect call transformation
4. Identical code folding
5. (Speculative) devirtualization
6. Constant propagation and function specialization (cloning)
7. Constructor/destructor merging
8. Inlining
9. Attribute discovery: pure, const, nothrow, nonfreeing, noreturn, malloc
10. Ipa-reference (collects list of static variables read/written by a given function)
11. Comdat optimizations
12. Points-to analysis (experimental Andresen-style, presently not scaling well to larger programs)
1. Symbol visibility optimizations
2. Unreachable function and variable removal
3. Inter-procedural profile propagation and indirect call transformation
4. Identical code folding
5. (Speculative) devirtualization
6. Constant propagation and function specialization (cloning)
7. Constructor/destructor merging
8. Inlining
9. Attribute discovery: pure, const, nothrow, nonfreeing, noreturn, malloc
10. Ipa-reference (collects list of static variables read/written by a given function)
Inter-procedural optimization passes in GCC

1. Symbol visibility optimizations
2. Unreachable function and variable removal
3. Inter-procedural profile propagation and indirect call transformation
4. Identical code folding
5. (Speculative) devirtualization
6. Constant propagation and function specialization (cloning)
7. Constructor/destructor merging
8. Inlining
9. Attribute discovery: pure, const, nothrow, nonfreeing, noreturn, malloc
10. Ipa-reference (collects list of static variables read/written by a given function)
11. comdat optimizations
Inter-procedural optimization passes in GCC

1. Symbol visibility optimizations
2. Unreachable function and variable removal
3. Inter-procedural profile propagation and indirect call transformation
4. Identical code folding
5. (Speculative) devirtualization
6. Constant propagation and function specialization (cloning)
7. Constructor/destructor merging
8. Inlining
9. Attribute discovery: pure, const, nothrow, nonfreeing, noreturn, malloc
10. Ipa-reference (collects list of static variables read/written by a given function)
11. comdat optimizations
12. Points-to analysis (experimental Andresen-style, presently not scaling well to larger programs)
Mod-ref is a new inter-procedural analysis pass collecting information about what memory locations can be modified or referenced by a given function. This improves quality of alias analysis and thus enables more optimization.
What is mod-ref pass

Mod-ref is a new inter-procedural analysis pass collecting information about what memory locations can be modified or referenced by a given function. This improves quality of alias analysis and thus enables more optimization.

Compile-time:

- Top-down propagation of mod-ref information to improve quality of early optimization passes
- Analysis for full IPA-propagation
- Streaming of summaries to LTO object files
What is mod-ref pass

Mod-ref is a new inter-procedural analysis pass collecting information about what memory locations can be modified or referenced by a given function. This improves quality of alias analysis and thus enables more optimization.

Compile-time:
- Top-down propagation of mod-ref information to improve quality of early optimization passes
- Analysis for full IPA-propagation
- Streaming of summaries to LTO object files

Link time (WPA):
- Streaming summaries in
- Inter-procedural propagation (dataflow based over strongly connected components of the callgraph)
- Streaming optimization summaries out
What is mod-ref pass

**Mod-ref** is a new inter-procedural analysis pass collecting information about what memory locations can be **modified** or **referenced** by a given function. This improves quality of alias analysis and thus enables more optimization.

**Compile-time:**

- Top-down propagation of mod-ref information to improve quality of early optimization passes
- Analysis for full IPA-propagation
- Streaming of summaries to LTO object files

**Link time (WPA):**

- Streaming summaries in
- Inter-procedural propagation (dataflow based over strongly connected components of the callgraph)
- Streaming optimization summaries out

**Link time (ltrans):**

- Streaming optimization summaries in
- Re-computation of mod-ref for optimized function bodies (after inlining) and top-down propagation
short *p,*q;
int a;

__attribute__((noinline))
void foo ()
{
*p=*q;
}

int test ()
{
    a = 1;
    foo ();
    return a;
}
short *p,*q;
int a;

__attribute__((noinline))
void foo ()
{
    *p=*q;
}

int test ()
{
    a = 1;
    foo ();
    return a;
}
short *p,*q;
int a;

__attribute__((noinline))
void foo ()
{
    *p=*q;
}

int test ()
{
    a = 1;
    foo ();
    return a;
}

GCC 11 code (with mod-ref):

foo:
movq q(%rip), %rax
movzwl (%rax), %edx
movq p(%rip), %rax
movw %dx, (%rax)
ret

test:
movl $1, a(%rip)
xorl %eax, %eax
call foo
-> movl $1, %eax
ret
short *p,*q;
int a;

__attribute__((noinline))
void foo ()
{
    *p=*q;
}

int test ()
{
    a = 1;
    foo ();
    return a;
}

Modref analysis dump:

- Analyzing load: q
  - Recording base_set=1 ref_set=1 parm=-1
- Analyzing load: p
  - Recording base_set=1 ref_set=1 parm=-1
- Analyzing load: *q.0_1
  - Recording base_set=2 ref_set=2 parm=-1
- Analyzing store: *p.1_2
  - Recording base_set=2 ref_set=2 parm=-1

Modref analysis dump:

loads:
  Base 0: alias set 1
  Ref 0: alias set 1
  Every access
  Base 1: alias set 2
  Ref 0: alias set 2
  Every access

stores:
  Base 0: alias set 2
  Ref 0: alias set 2
  Every access
Access trees: tracking access ranges

```c
__attribute__((noinline))
void foo (short *p, short *q)
{
    p[1]=q[2];
}

int test (short *p, short *q)
{
    p[2]=1;
    foo (p,q);
    return p[2];
}
```
__attribute__((noinline))

void foo (short *p, short *q)
{
    p[1]=q[2];
}

int test (short *p, short *q)
{
    p[2]=1;
    foo (p,q);
    return p[2];
}
__attribute__((noinline))
void foo (short *p, short *q)
{
    p[1]=q[2];
}

int test (short *p, short *q)
{
    p[2]=1;
    foo (p,q);
    return p[2];
}

GCC 11 code (with mod-ref):

foo:
    movzwl 4(%rsi), %eax
    movw %ax, 2(%rdi)
    ret

test:
    movl $1, %eax
    movw %ax, 4(%rdi)
    call foo
    movl $1, %eax
    ret
__attribute__((noinline))

void foo (short *p, short *q)
{
    p[1] = q[2];
}

int test (short *p, short *q)
{
    p[2] = 1;
    foo (p, q);
    return p[2];
}

- Analyzing load: MEM[(short int *)q_4(D) + 4B]
  - Recording base_set=1 ref_set=1 parm=1
- Analyzing store: MEM[(short int *)p_1(D) + 4B]
  - Recording base_set=1 ref_set=1 parm=0
loads:
stores:
    Base 0: alias set 1
    Ref 0: alias set 1
access: Parm 0 param offset:4 offset:0 size:16 max_size:16

GCC 11 code (with mod-ref):

foo:
    movzwl 4(%rsi), %eax
    movw %ax, 2(%rdi)
    ret

test:
    movl $1, %eax
    movw %ax, 4(%rdi)
    call foo
    movl $1, %eax
    ret
void bar (void);

__attribute__((noinline))
void foo (short *p)
{
    *p=1;
}

int test ()
{
    short p;
    foo (&p);
    p=2;
    bar ();
    return p;
}
void bar (void);

__attribute__ ((noinline))
void foo (short *p)
{
    *p=1;
}

int test ()
{
    short p;
    foo (&p);
    p=2;
    bar ();
    return p;
}
void bar (void);

__attribute__((noinline))
void foo (short *p)
{
    *p = 1;
}

int test ()
{
    short p;
    foo (&p);
    p = 2;
    bar ();
    return p;
}

GCC 11 code (with mod-ref):

foo:
    movl $1, %eax
    movw %ax, (%rdi)
    ret

test:
    subq $24, %rsp
    leaq 14(%rsp), %rdi
    call foo
    call bar
    movl $2, %eax
    addq $24, %rsp
    ret
void bar (void);

__attribute__((noinline)) void foo (short *p)
{
   *p=1;
}

int test ()
{
   short p;
   foo (&p);
   p=2;
   bar ();
   return p;
}

Analyzing flags of ssa name: p_2(D)
Analyzing stmt: *p_2(D) = 1;
current flags of p_2(D) direct noescape nodirectescape
parm 0 flags: direct noescape nodirectescape

GCC 11 code (with mod-ref):

foo:

   movl $1, %eax
   movw %ax, (%rdi)
   ret

test:

   subq $24, %rsp
   leaq 14(%rsp), %rdi
   call foo
   call bar

   movl $2, %eax
   addq $24, %rsp
   ret
void bar (void);

__attribute__((noinline))
void foo (short *p)
{
    *p = 1;
}

int test ()
{
    short p;
    foo (&p);
    p = 2;
    bar ();
    return p;
}

Analyzing flags of ssa name: p_2(D)
Analyzing stmt: *p_2(D) = 1;
current flags of p_2(D) direct noescape nodirectescape

d parm 0 flags: direct noescape nodirectescape

Noescape flag: memory pointed-to by argument does not escape to global memory or other parameters
void bar (void);
short *globalptr;
__attribute__ ((noinline))
void foo (short **p)
{
    globalptr = *p;
}
int test ()
{
    short p, *q=&p;
    foo (&q);
    q=&p;
    bar ()
    return q==&p;
}
void bar (void);
short *globalptr;
__attribute__((noinline))
void foo (short **p)
{
    globalptr = *p;
}
int test ()
{
    short p, *q=&p;
    foo (&q);
    q=&p;
    bar ();
    return q==&p;
}
void bar (void);
short *globalptr;
__attribute__ ((noinline))
void foo (short **p)
{
    globalptr = *p;
}
int test ()
{
    short p, *q=&p;
    foo (&q);
    q=&p;
    bar ();
    return q==&p;
}

parm 0 flags: nodirectescape

Nodirectescape flag: object pointed-to by argument does not escape to global memory or other parameters. However memory pointed-to indirectly may escape.

GCC 11 code (with mod-ref):

foo:
    movl $1, %eax
    movw %ax, (%rdi)
    ret
test:
    subq $24, %rsp
    leaq 14(%rsp), %rdi
    call foo
    call bar
    ->
    movl $2, %eax
    addq $24, %rsp
    ret
void bar (void);
short global;

__attribute__((noinline))
void foo (short **p)
{
    global = (*p != 0);
}

int test ()
{
    short p=1234, *q=&p;
    foo (&q);
    bar ();
    return p;
}"
void bar (void);
short global;

__attribute__((noinline))
void foo (short **p)
{
    global = (*p != 0);
}

int test ()
{
    short p=1234, *q=&p;
    foo (&q);
    bar ();
    return p;
}

GCC 11 code (with mod-ref):

foo:
.LFB0:
.cfi_startproc
    xorl %eax, %eax
    cmpq $0, (%rdi)
    setne %al
    movw %ax, global(%rip)
    ret

test:
    subq $24, %rsp
    leaq 6(%rsp), %rax
    leaq 8(%rsp), %rdi
    movq %rax, 8(%rsp)
    call foo
    call bar
    movl $1234, %eax
    addq $24, %rsp
    ret

Direct: Memory pointed-to indirectly by the argument is never accessed.
void bar (void);
short global;

__attribute__((noinline))
void foo (short **p)
{
    global = (*p != 0);
}

int test ()
{
    short p=1234, *q=&p;
    foo (&q);
    bar ();
    return p;
}

**Direct:** Memory pointed-to indirectly by the argument is never accessed.
int a, b, c;

__attribute__((noinline))
void foo (short *p)
{
    c = (*p != 0);
}

int test ()
{
    int *p = c ? &a : &b;
    foo (&q);
    c = 1;
    *p = 2;
    return c;
}
int a, b, c;

__attribute__((noinline))
void foo (short *p)
{
    c = (*p != 0);
}

test()
{
    int *p = c ? &a : &b;
    foo (&q);
    c = 1;
    *p = 2;
    return c;
}
int a, b, c;

__attribute__((noinline))
void foo (short *p)
{
    c = (*p != 0);
}

int test ()
{
    int *p = c ? &a : &b;
    foo (&q);
    c = 1;
    *p = 2;
    return c;
}

clobber: parameter is read only.

GCC 11 code (with mod-ref):

test:
    subq $16, %rsp
    movl c(%rip), %eax
    movl $a, %edx
    leaq 8(%rsp), %rdi
    testl %eax, %eax
    movl $b, %eax
    cmove %rax, %rdx
    movq %rdx, 8(%rsp)
    call foo
    movl $2, (%rdx)
    movl $1, %eax
    movl $1, c(%rip)
    addq $16, %rsp
Mod-ref feature summary

1. Load/stores access trees tracking alias sets, bases, offsets and sizes of individual accesses in the function

2. Discovery of EAF_NOESCAPE, EAF_NODIRECTESCAPE, EAF_DIRECT, EAF_NOTCLOBBER

3. New in GCC 12: Discovery of EAF_NOT_RETURNED, EAF_NOT_READ, EAF_UNUSED

All analysis except for discovery of EAF_NOT_RETURNED is done for whole program. EAF_NOT_RETURNED is propagated within one translation unit only due to lack of infrastructure for return functions.
21% increase in disambiguation rate for cc1plus LTO build.
cc1plus disambiguations (by oracle)

11% more TBAA, 1312% more PTA
Benchmarks (Zen CPU)

modref:
- 0.68% improvement for SPECint2017 -O2 -flto
- 0.92% improvement for SPECfp2017 -O2 -flto

strict aliasing:
- 1.64% improvement for SPECint2017 -O2 -flto
- 1.31% improvement for SPECfp2017 -O2 -flto

PTA:
- 0.14% regression for SPECint2017 -O2 -flto
- 1.83% improvement for SPECfp2017 -O2 -flto

all together:
- 2.11% improvement for SPECint2017 -O2 -flto
- 3.91% improvement for SPECfp2017 -O2 -flto
Some non-spec benchmarks

- 21.43% improvement for assignment (nbench)
- 11.83% improvement for fatigue (polyhedron)
- 2.52% improvement for mdbx (polyhedron)
- 2.23% improvement for ac (polyhedron)
- 7.11% regression for ac (polyhedron)
- 7.12% regression for lu decomposition (nbench)