Interprodecural optimizations in GCC

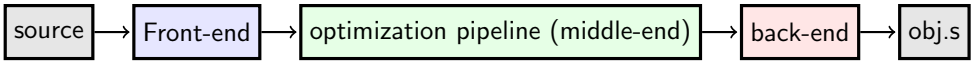
Martin Jambor, Jan Hubička, Martin Liška

SUSE Labs

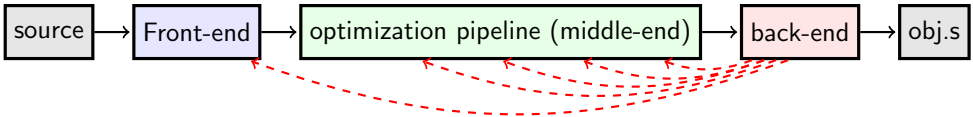
October 5th, 2019

Introduction to interprocedural optimizations

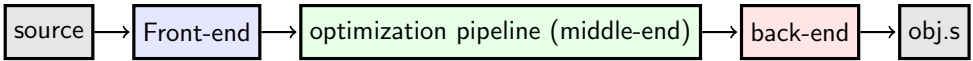
Optimization pipeline



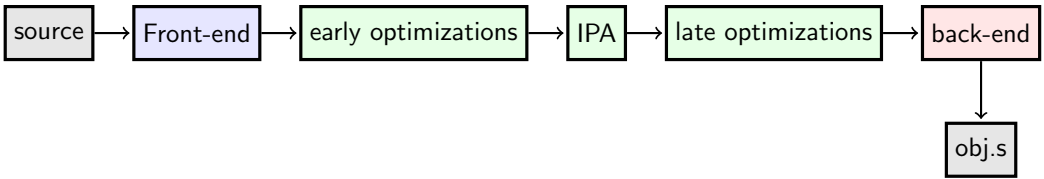
Optimization pipeline



Optimization pipeline

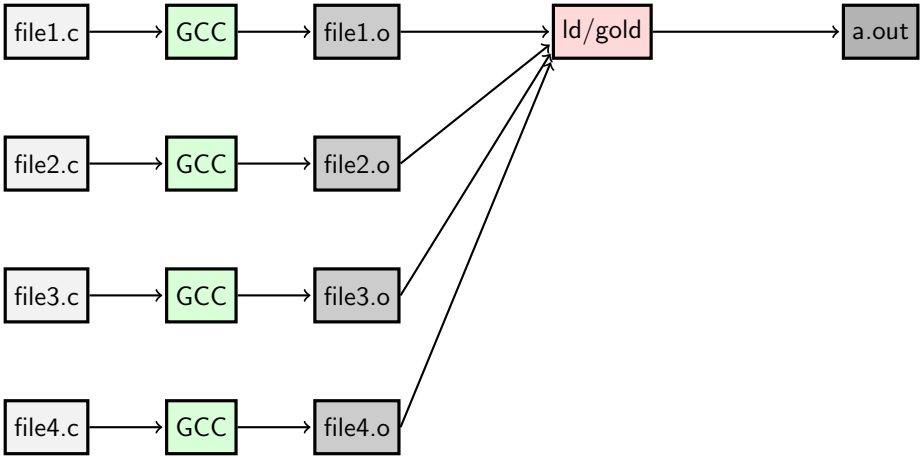


Optimization pipeline

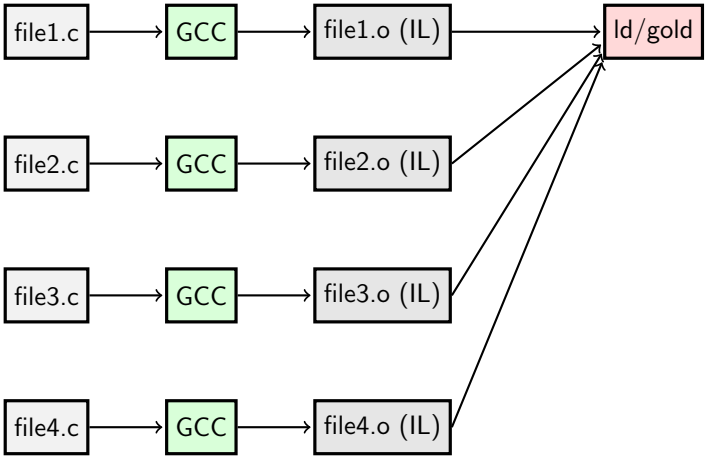


Link-time Optimization

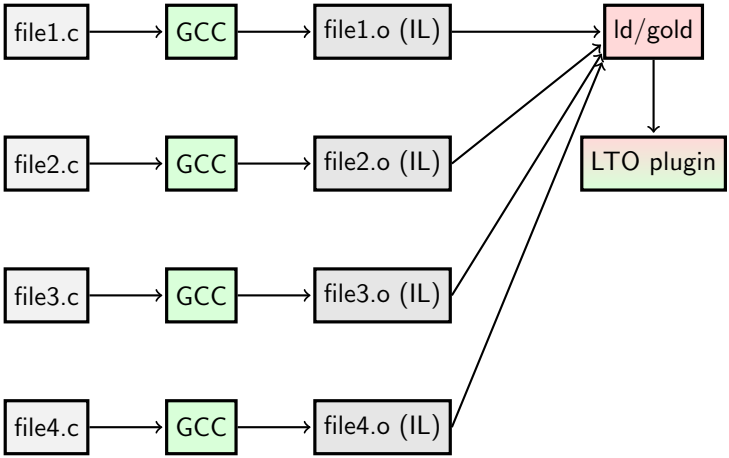
Traditional (non-LTO) build



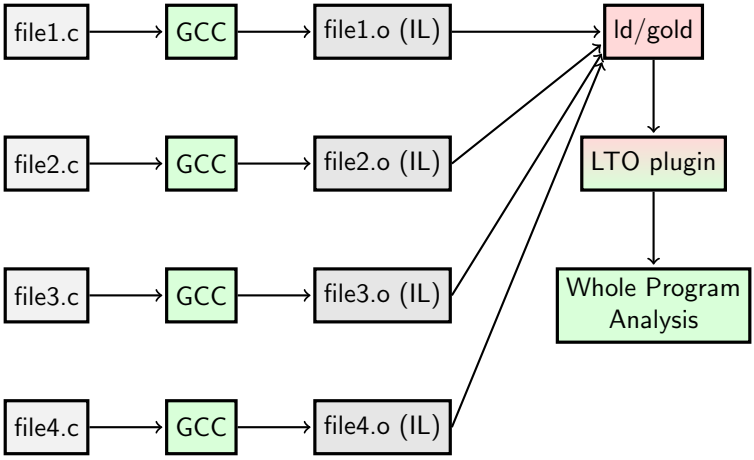
LTO


file1.c
GCC
file1.o (IL)
ld/gold
file2.c
GCC
file2.o (IL)
file3.c
GCC
file3.o (IL)
file4.c
GCC
file4.o (IL)

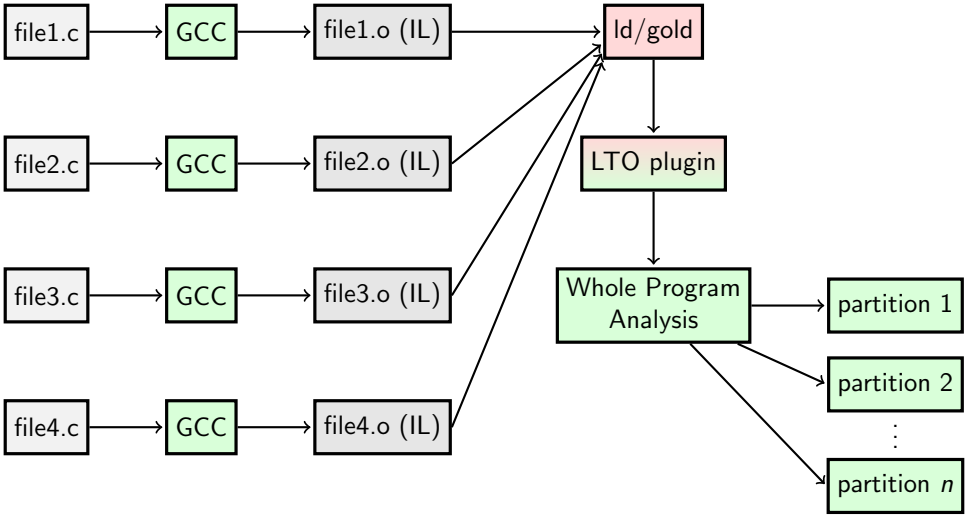
LTO


file1.c
GCC
file1.o (IL)
ld/gold
file2.c
GCC
file2.o (IL)
LTO plugin
file3.c
GCC
file3.o (IL)
file4.c
GCC
file4.o (IL)

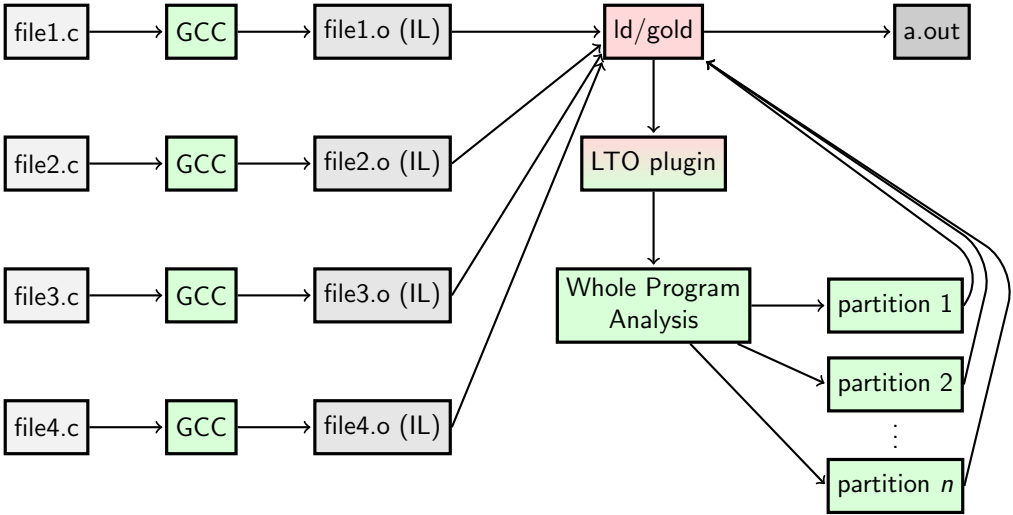
LTO


file1.c
GCC
file1.o (IL)
ld/gold
file2.c
GCC
file2.o (IL)
LTO plugin
file3.c
GCC
file3.o (IL)
Whole Program Analysis
file4.c
GCC
file4.o (IL)

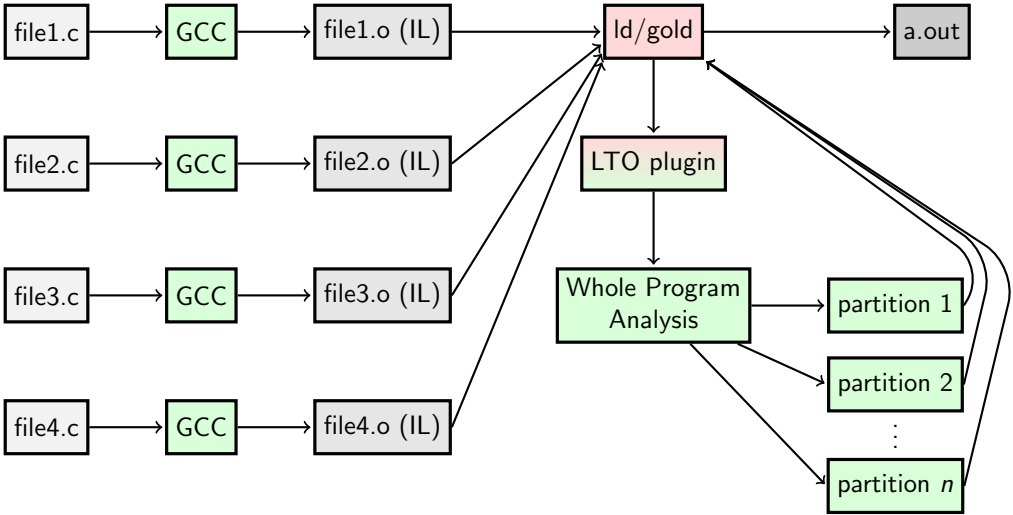
LTO


file1.c
GCC
file1.o (IL)
ld/gold
file2.c
GCC
file2.o (IL)
LTO plugin
file3.c
GCC
file3.o (IL)
Whole Program Analysis
partition 1
partition 2
partition n
file4.c
GCC
file4.o (IL)

LTO

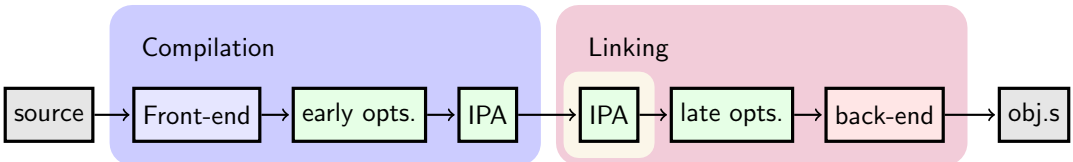

file1.c
GCC
file1.o (IL)
ld/gold
a.out
file2.c
GCC
file2.o (IL)
LTO plugin
file3.c
GCC
file3.o (IL)
Whole Program Analysis
partition 1
partition 2
file4.c
GCC
file4.o (IL)
partition n

LTO



...this is how the vast majority of openSUSE Tumbleweed packages are built.

Compiler structure one more time



Symbol promotion

- Many symbols which without LTO have external linkage are with LTO local.

Symbol promotion

- Many symbols which without LTO have external linkage are with LTO local.
  In C-speak, **stuff (variables, functions) magically becomes static**.

Symbol promotion

- Many symbols which without LTO have external linkage are with LTO local. In C-speak, **stuff (variables, functions) magically becomes static**.
- This means we have total control over the symbol.

Symbol promotion

- Many symbols which without LTO have external linkage are with LTO local.
  In C-speak, **stuff (variables, functions) magically becomes static**.
- This means we have total control over the symbol.
- Performed with precise information from linker.
  - Even C++ COMDAT symbols can be privatized if their address does not escape.
  - Often simplifies dynamic linking.

Let's look at some IPA optimizations

Unreachable code removal

```
for (int i = 0; i < LIMIT; i++)
  {
    /* ... */
    if (i > 64)
      handle_big_index (i);
    /* ... */

  }
```

```
static void
handle_big_index (int i)
{
  /* ... */
}
```

Unreachable code removal

```
for (int i = 0; i < LIMIT; i++)
  {
    /* ... */
    if (i > 64)
      handle_big_index (i);
    /* ... */

  }
```

```
void
handle_big_index (int i)
{
  /* ... */
}
```

Unreachable code removal

```
for (int i = 0; i < LIMIT; i++)
  {
    /* ... */
    if (i > 64)
      handle_big_index (i);
    /* ... */

  }
```

```
void
handle_big_index (int i)
{
  /* ... */
}
```

- Particularly important for C++ template-heavy input.

Unreachable code removal

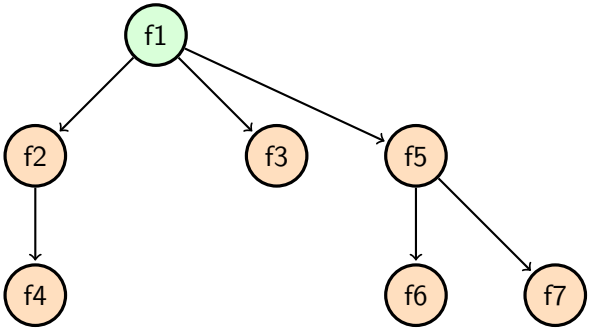
```
for (int i = 0; i < LIMIT; i++)
  {
    /* ... */
    if (i > 64)
      handle_big_index (i);
    /* ... */

  }
```

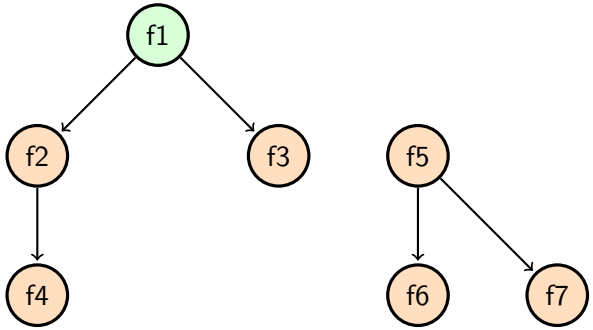
```
void
handle_big_index (int i)
{
  /* ... */
}
```

- Particularly important for C++ template-heavy input.
- Text size of 510.parest_r drops to about 20% of non-LTO build.
- Text size of Firefox is approximately 89% of non-LTO build.

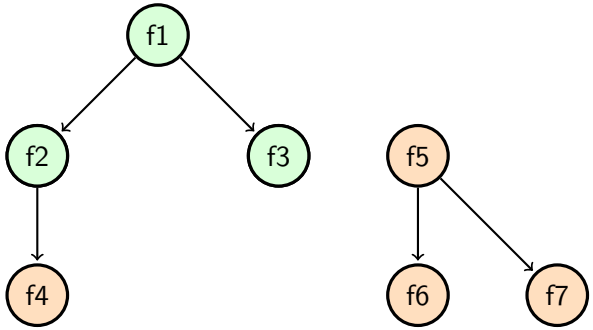
Unreachable code removal (2)



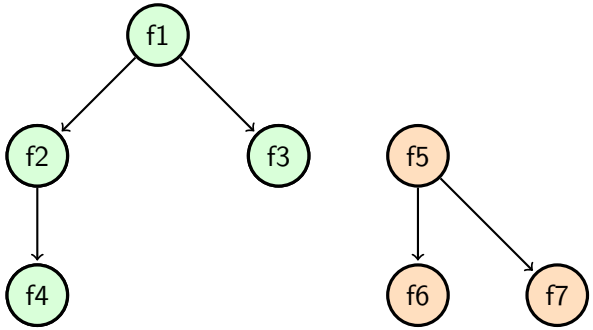
Unreachable code removal (2)



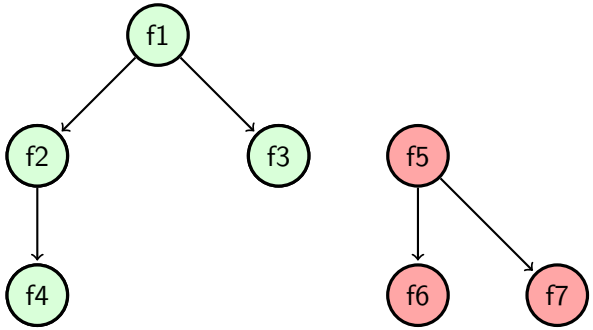
Unreachable code removal (2)



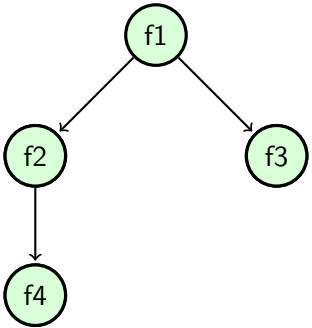
Unreachable code removal (2)



Unreachable code removal (2)



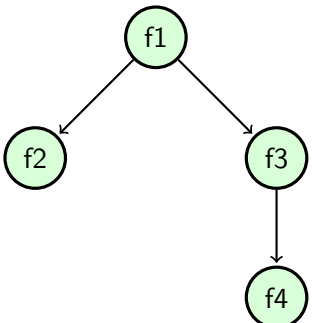
Unreachable code removal (2)



Attribute discovery

Automatic discovery of:

- pure, const, malloc, noreturn,
- nothrow (can save 46% EH info when LTO-building Firefox, which saves a lot of code in cleanup regions.



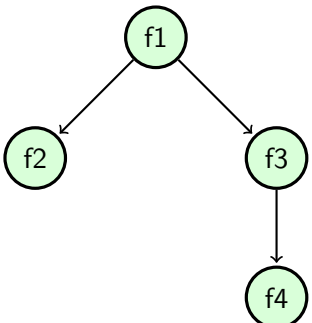
Attribute discovery

Automatic discovery of:

- pure, const, malloc, noreturn,
- nothrow (can save 46% EH info when LTO-building Firefox, which saves a lot of code in cleanup regions.

Results available to user:

- `-Wsuggest-attribute`



Attribute discovery

Automatic discovery of:

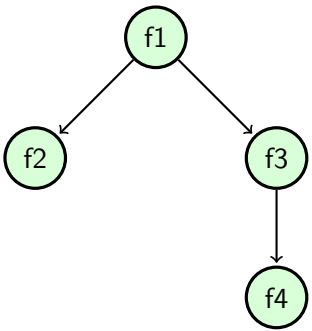
- pure, const, malloc, noreturn,
- nothrow (can save 46% EH info when LTO-building Firefox, which saves a lot of code in cleanup regions.

Results available to user:

- `-Wsuggest-attribute`

Similar propagation of profile information:

- Regions calling cold code are also cold



Identical Code Folding (ICF)

```
static int bar1(int v)
{
  return v + 2;
}
```

```
static int bar2(int v)
{
  return v + 2;
}
```

Identical Code Folding (ICF)

```
static int bar1(int v)
{
  return v + 2;
}

int foo1(int a)
{
  return a * bar1(a);
}
```

```
static int bar2(int v)
{
  return v + 2;
}

int foo2(int a)
{
  return a * bar2(a);
}
```

Identical Code Folding (ICF)

```
static int bar1(int v)
{
  return v + 2;
}


int foo1(int a)
{
  return a * bar1(a);
}
```

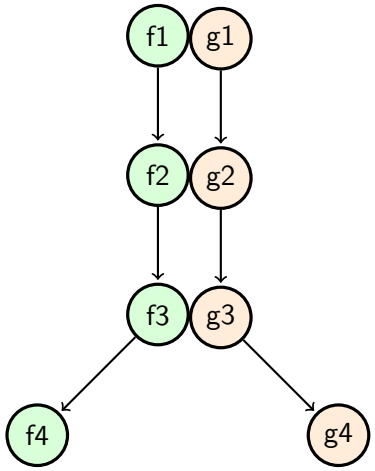
```
static int bar2(int v)
{
  return v + 2;
}


int foo2(int a)
{
  return a * bar2(a);
}
```

```
foo2:
        .cfi_startproc
        jmp     foo1
        .cfi_endproc
```

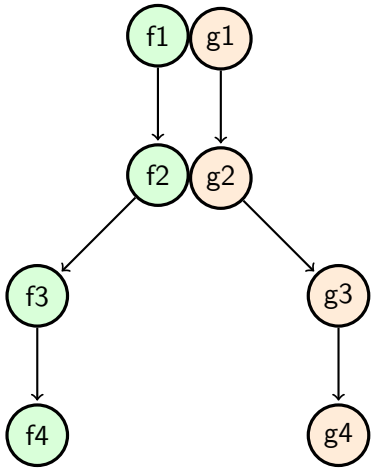
ICF (2)

- Calculate a fingerprint (hash) of each function.
- If two functions have the same hash, compare their bodies
- Check that they call semantically identical functions and if not, declare them different and propagate the information as appropriate.



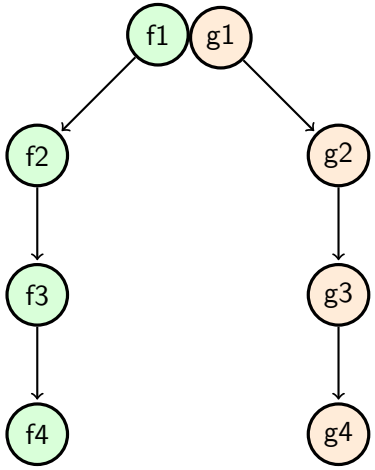
ICF (2)

- Calculate a fingerprint (hash) of each function.
- If two functions have the same hash, compare their bodies
- Check that they call semantically identical functions and if not, declare them different and propagate the information as appropriate.



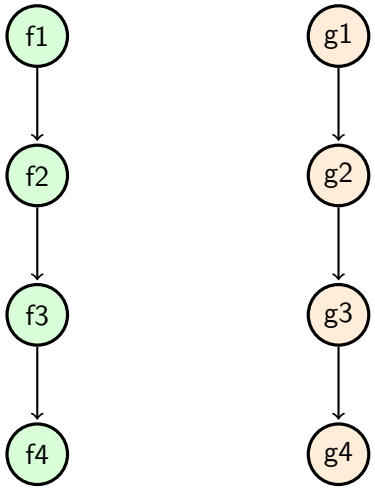
ICF (2)

- Calculate a fingerprint (hash) of each function.
- If two functions have the same hash, compare their bodies
- Check that they call semantically identical functions and if not, declare them different and propagate the information as appropriate.



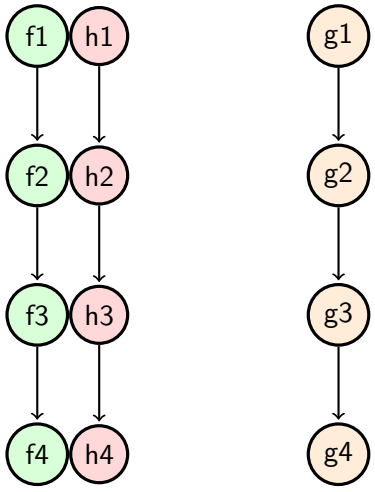
ICF (2)

- Calculate a fingerprint (hash) of each function.
- If two functions have the same hash, compare their bodies
- Check that they call semantically identical functions and if not, declare them different and propagate the information as appropriate.

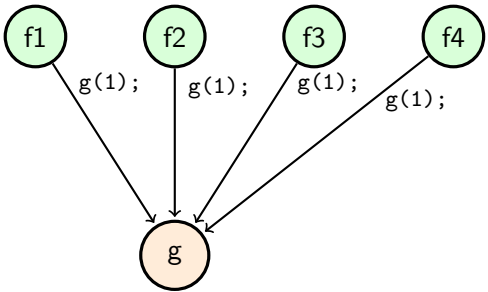


ICF (2)

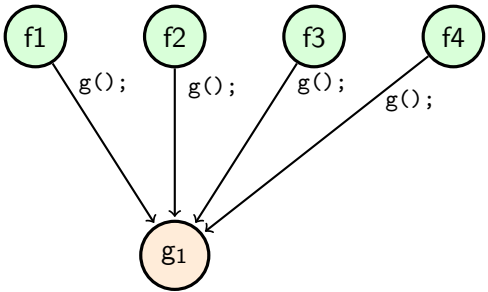
- Calculate a fingerprint (hash) of each function.
- If two functions have the same hash, compare their bodies
- Check that they call semantically identical functions and if not, declare them different and propagate the information as appropriate.



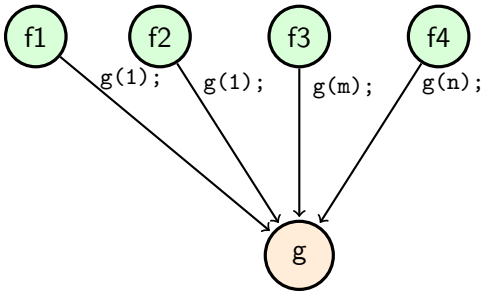
Propagation of “stuff” (IPA-CP)



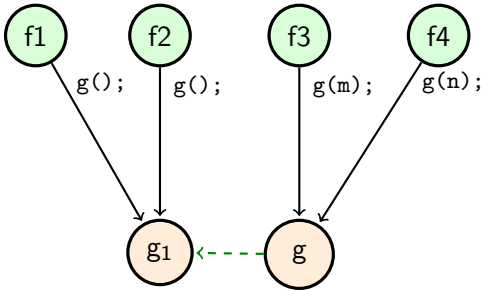
Propagation of “stuff” (IPA-CP)



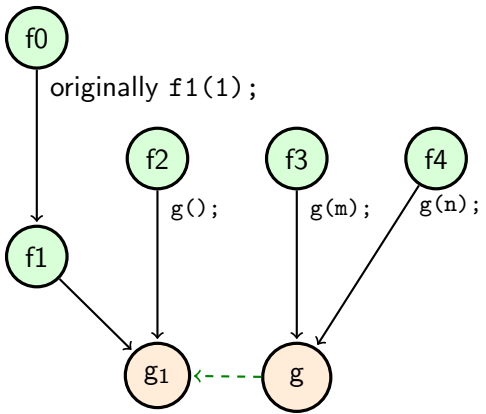
Propagation of “stuff” (IPA-CP)



Propagation of “stuff” (IPA-CP)



Propagation of “stuff” (IPA-CP)



IPA-CP (2)

IPA-CP tracks and propagates:

- constants,
- constants passed by reference and/or as part of bigger aggregates (discovery limited to simple cases),
- polymorphic call context for devirtualization (and speculative devirtualization),
- value-ranges
- which bits can be non-zero.

IPA-CP (3)

Phases of IPA-CP:

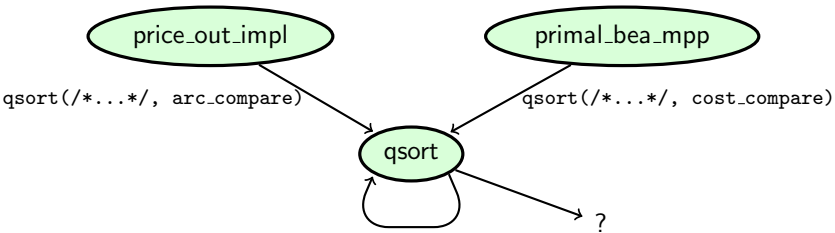
1. Summarize behavior of functions:
   - What are the properties of parameters that they pass to functions they call?
   - How they pass on values they received in formal parameters in actual arguments of their calls?
   - How do specific values of parameters affect this function?

IPA-CP (3)

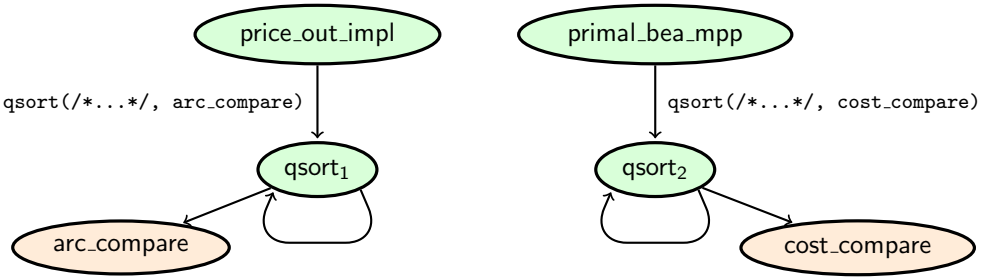
Phases of IPA-CP:

1. Summarize behavior of functions:
   - What are the properties of parameters that they pass to functions they call?
   - How they pass on values they received in formal parameters in actual arguments of their calls?
   - How do specific values of parameters affect this function?
2. Propagate known values (and other stuff) from callers to callees (top to bottom – iterate in recursive regions).
3. Propagate effect estimations from callees to callers (bottom to top).
4. Traverse the call graph and make cloning decisions, save results.

IPA-CP (4)


price_out_impl
primal_bea_mpp
qsort(/*...*/, arc_compare)
qsort(/*...*/, cost_compare)
qsort
?

IPA-CP (4)


price_out_impl
qsort(/*...*/, arc_compare)
qsort1
arc_compare
primal_bea_mpp
qsort(/*...*/, cost_compare)
qsort2
cost_compare

Inlining

Inlining:

- Greedy algorithm, picking the best inlining candidate as long as it can (i.e. there are no more candidates or growth limits are reached).

Inlining

Inlining:

- Greedy algorithm, picking the best inlining candidate as long as it can (i.e. there are no more candidates or growth limits are reached).
- Call context effects information used (the same we have for IPA-CP).

Inlining

Inlining:

- Greedy algorithm, picking the best inlining candidate as long as it can (i.e. there are no more candidates or growth limits are reached).
- Call context effects information used (the same we have for IPA-CP).
- Inlining metrics:
  (saved exe. time * call frequency) / (function growth * MAX(1, total growth))

Inlining

Inlining:

- Greedy algorithm, picking the best inlining candidate as long as it can (i.e. there are no more candidates or growth limits are reached).
- Call context effects information used (the same we have for IPA-CP).
- Inlining metrics:
  (saved exe. time * call frequency) / (function growth * MAX(1, total growth))

  \+ special points for:
  - making a call direct (devirtualization)
  - newly constant iteration count
  - newly constant stride

Inlining

Inlining:

- Greedy algorithm, picking the best inlining candidate as long as it can (i.e. there are no more candidates or growth limits are reached).
- Call context effects information used (the same we have for IPA-CP).
- Inlining metrics:
  (saved exe. time * call frequency) / (function growth * MAX(1, total growth))

  \+ special points for:
  - making a call direct (devirtualization)
  - newly constant iteration count
  - newly constant stride

  \- penalties for:
  - recursive calls
  - cross-module inlining

Inlining (2)

Size limit for inlining candidates:

- Softer for functions marked inline,
- and for functions which we estimate will be sped up by over 15%.

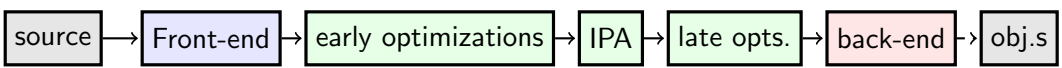
Inlining (2)

Size limit for inlining candidates:

- Softer for functions marked inline,
- and for functions which we estimate will be sped up by over 15%.

Early inlining:

- Small pass inlining all calls which are always beneficial.
- Important to remove (C++) levels of abstraction early



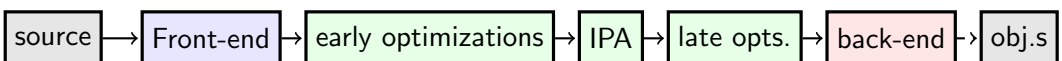
Inlining (2)

Size limit for inlining candidates:

- Softer for functions marked inline,
- and for functions which we estimate will be sped up by over 15%.

Early inlining:

- Small pass inlining all calls which are always beneficial.
- Important to remove (C++) levels of abstraction early



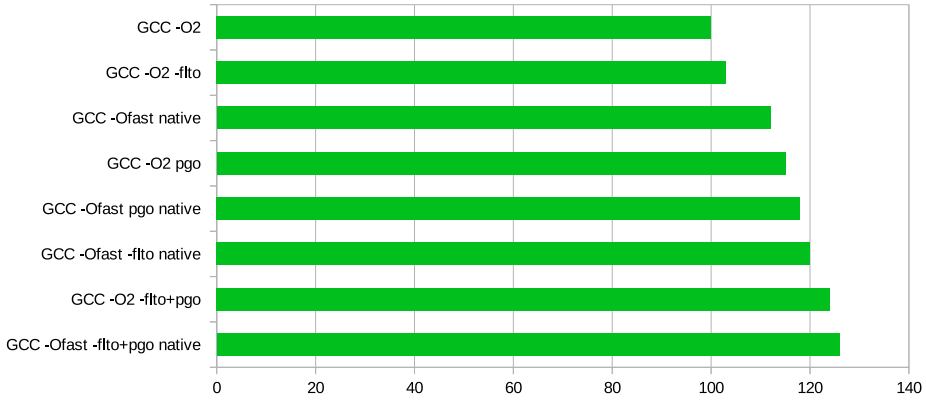
Inlining in big programs is hard

- `-Winline`
- `-fopt-info-inline`

Effect of LTO and PGO

GCC trunk on July 31st on an AMD Zen2 machine:

(SPEC score, higher is better)



Removing unused parameters

```
static bool
vectorizable_simd_clone_call (stmt_vec_info stmt_info,
                              gimple_stmt_iterator *gsi,
                              stmt_vec_info *vec_stmt, slp_tree slp_node,
                              stmt_vector_for_cost *)
```

Removing unused parameters (2)

```
void
backprop::optimize_builtin_call (gcall *call, tree lhs, const usage_info *info)
{
  /* If we have an f such that -f(x) = f(-x), and if the sign of the result
     doesn't matter, strip any sign operations from the input.*/
  if (info->flags.ignore_sign
      && negate_mathfn_p (gimple_call_combined_fn (call)))
    {
      tree new_arg = strip_sign_op (gimple_call_arg (call, 0));
      if (new_arg)
        {
          prepare_change (lhs);
          gimple_call_set_arg (call, 0, new_arg);
          complete_change (call);
        }
    }
}
```

Removing unused parameters (3)

```
static void
get_constraint_for_component_ref (tree t, vec<ce_s> *results,
                                  bool address_p, bool lhs_p)
```

Removing unused parameters (3)

```
static void
get_constraint_for_component_ref (tree t, vec<ce_s> *results,
                                  bool address_p, bool lhs_p)
{
 /* ... */
   get_constraint_for_1 (TREE_OPERAND(t, 0), results, false, lhs_p);
 /* ... */
}
```

Removing unused parameters (3)

```
static void
get_constraint_for_component_ref (tree t, vec<ce_s> *results,
                                  bool address_p, bool lhs_p)
{
 /* ... */
  get_constraint_for_1 (TREE_OPERAND(t, 0), results, false, lhs_p);
 /* ... */
}
static void
get_constraint_for_1 (tree t, vec<ce_s> *results, bool address_p,
                      bool lhs_p) {
 /* ... */
get_constraint_for_component_ref (t, results, address_p, lhs_p);
 /* ... */
get_constraint_for_1 (val, &tmp, address_p, lhs_p);
 /* ... */
}
```

Removing unused return values

```
/* Visit and value number STMT, return true if the value number
   changed.  */
static bool
visit_stmt (gimple *stmt, bool backedges_varying_p = false)
{
  bool changed = false;
 /* ...  */
    changed = defs_to_varying (stmt);
 /* ...  */
        changed = visit_reference_op_store (lhs, rhs1, ass);
 /* ...  */
                changed = visit_reference_op_load (lhs, rhs1, ass);
 /* ...  */
        changed = visit_reference_op_load (lhs, rhs1, ass);
 /* ...  */
 done:
  return changed;
}
```

Removing unused return values (2)

Per function, there are two flags:

- Can we change the signature of this function?
- Does this function return value?

Removing unused return values (2)

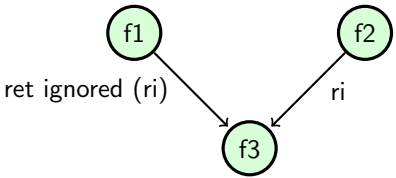
Per function, there are two flags:

- Can we change the signature of this function?
- Does this function return value?

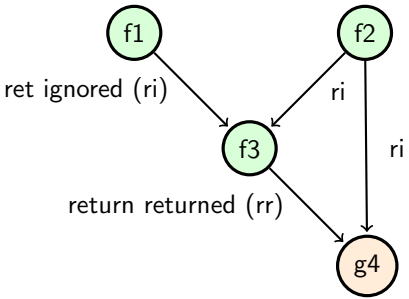
Per call, there are also two flags:

- Is the return value ignored?
- If not, is it only used to construct return value of the caller?

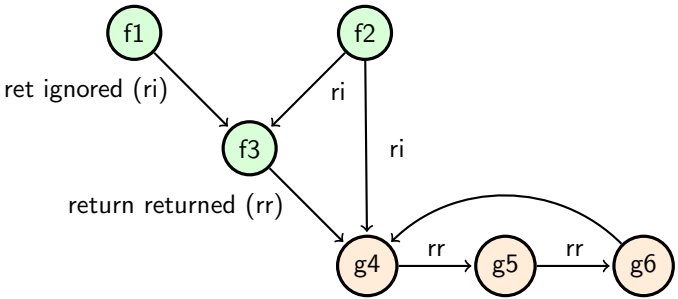
Removing unused return values (3)



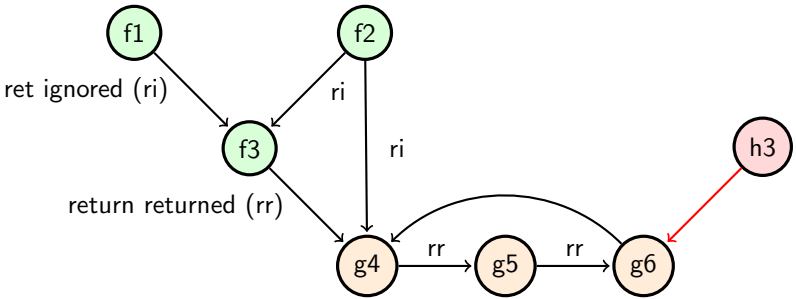
Removing unused return values (3)



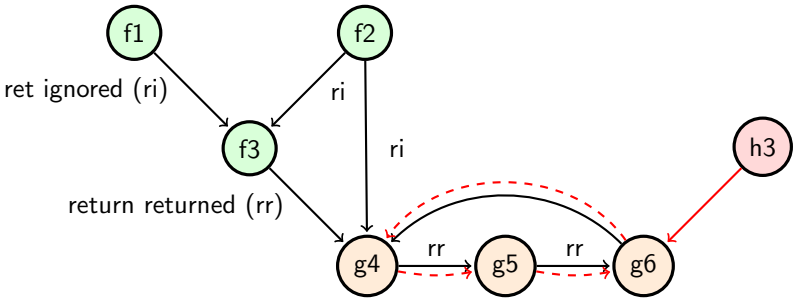
Removing unused return values (3)



Removing unused return values (3)



Removing unused return values (3)



Splitting an aggregate parameter

```
struct point
{
  unsigned h,s,l;
};
```

```
static unsigned
int process_l (point p)
{
  unsigned val = p.l;
  /* ... */
}

foo (pt);
```

→

```
static unsigned
int process_l (unsigned l)
{
  unsigned val = l;
  /* ... */
}

foo (pt.l);
```

Splitting an aggregate parameter (2)

```
struct point
{
  unsigned h,s,l;
};
```

```
static unsigned
int process_l (point *p)
{
  unsigned val = p->l;
  /* ... */
}

foo (ptr_pt);
```

→

```
static unsigned
int process_l (unsigned l)
{
  unsigned val = l;
   /* ... */
}

foo (ptr_pt->l);
```

Thank you