Optimizing C++ std::vector use in GCC 14

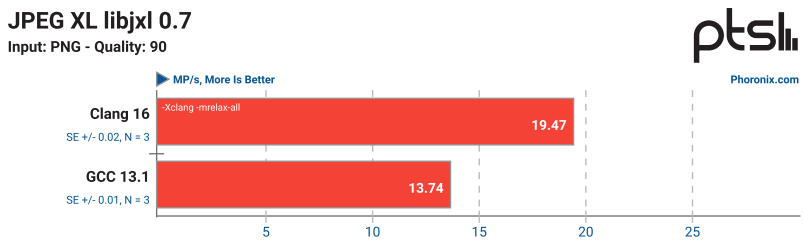
Jan Hubička, Martin Jambor

SUSE Labs

April 23, 2024

LLVM Clang 16 vs. GCC 13 Compiler Performance On Intel Raptor Lake

Written by Michael Larabel in Software on 11 May 2023 at 11:00 AM EDT.
Page 3 of 5. 35 Comments.

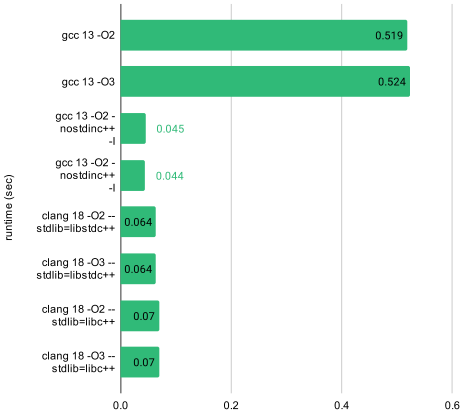


1. (CXX) g++ options: -O3 -march=native -flto -fno-rtti -funwind-tables -O2 -fPIE -pie -latomic

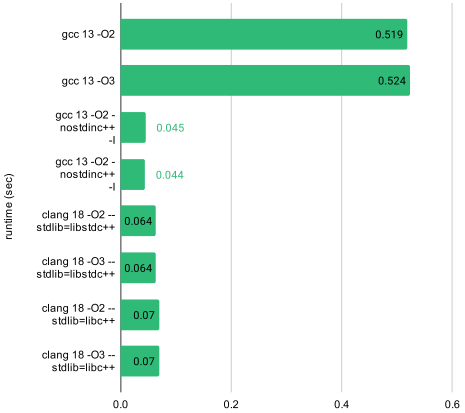
Simple testcase

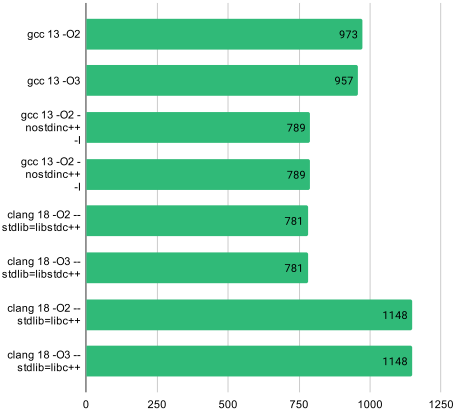
```
typedef std::pair<uint32_t, uint32_t> pair_t;
pair_t pair;

void test()
{
  std::vector <pair_t> stack;
  stack.push_back (pair);
  while (!stack.empty())
    {
      pair_t cur = stack.back();
      stack.pop_back();
      if (!cur.first)
        {
          cur.second++;
          stack.push_back (cur);
        }
      if (cur.second > 10000)
        break;
    }
}
```

Benchmark runtime and size (GCC vs Clang; libstdc++ vs libc++)

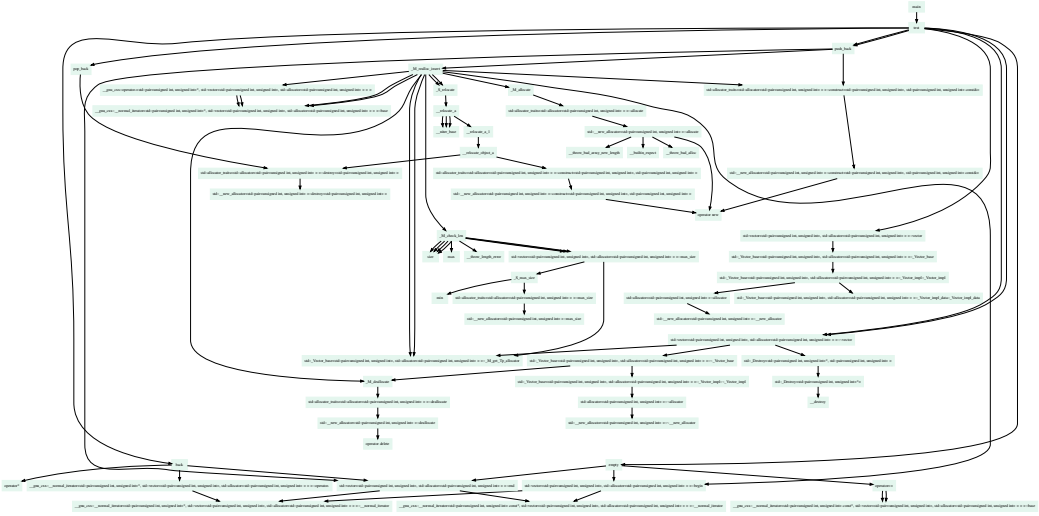


Benchmark runtime and size (GCC vs Clang; libstdc++ vs libc++)

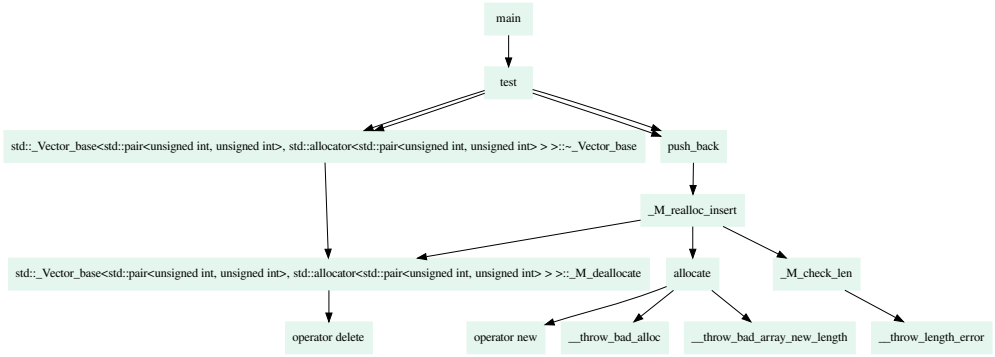




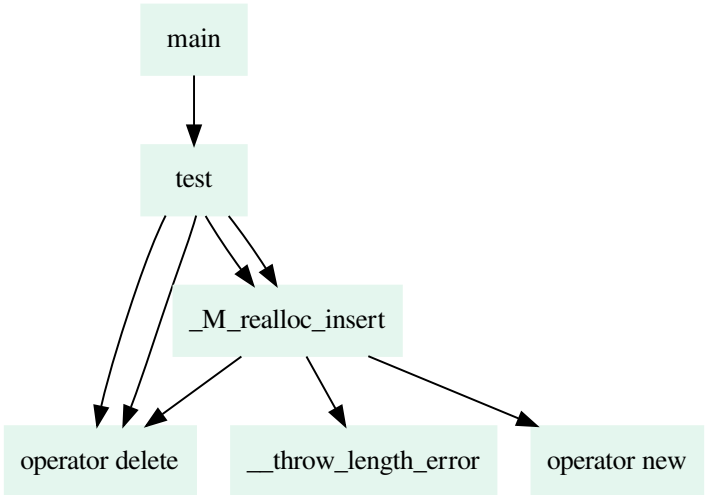
Call graph



Call graph after early inlining and optimization



Call graph after interprocedural optimization



Why `_M_realloc_insert` is not inlined?

1. 376 bytes of x86-64 assembly
2. 111 instructions

Why `_M_realloc_insert` is not inlined?

1. 376 bytes of x86-64 assembly
2. 111 instructions
3. After early optimization GCC thinks it will be 48 "instructions"
4. Inline limit: 15 for -O2, 30 for -O3

Why `_M_realloc_insert` is not inlined?

1. 376 bytes of x86-64 assembly
2. 111 instructions
3. After early optimization GCC thinks it will be 48 "instructions"
4. Inline limit: 15 for -O2, 30 for -O3
5. Declaring it inline bumps limit to 70 for -O2, 200 for -O3

Why `_M_realloc_insert` is not inlined?

1. 376 bytes of x86-64 assembly
2. 111 instructions
3. After early optimization GCC thinks it will be 48 "instructions"
4. Inline limit: 15 for -O2, 30 for -O3
5. Declaring it inline bumps limit to 70 for -O2, 200 for -O3

Declaring `_M_realloc_insert` inline works, but bloats code.

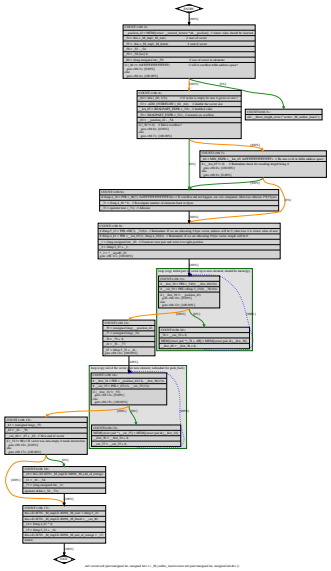
(This is what Clang's libc++ does)

Why `_M_realloc_insert` is not inlined?

1. 376 bytes of x86-64 assembly
2. 111 instructions
3. After early optimization GCC thinks it will be 48 “instructions”
4. Inline limit: 15 for -O2, 30 for -O3
5. Declaring it inline bumps limit to 70 for -O2, 200 for -O3

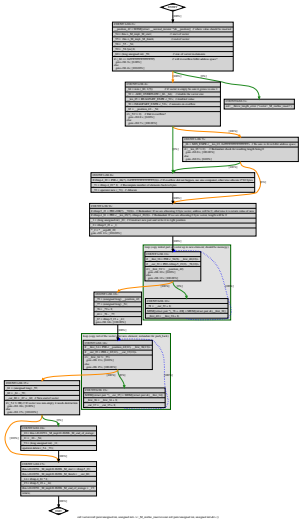
Declaring `_M_realloc_insert` inline works, but bloats code.

(This is what Clang's libc++ does)

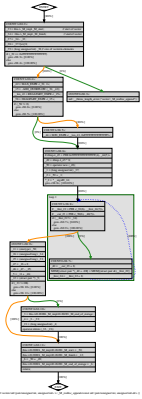


Optimizing `_M_realloc_insert`

`_M_realloc_insert` (GCC 13)



`_M_realloc_append` (GCC 14)



376 → 257 bytes (46% less)
111 → 76 instructions (46% less)

(Clang-18 472; 99 instructions)

PR109849 suboptimal code for vector walking loop

Some commits solving this problem:

- optimize `std::vector::push_back`
- Use `memcpy` instead of `memmove` in `__relocate_a_1`

PR109849 suboptimal code for vector walking loop

Some commits solving this problem:

- optimize `std::vector::push_back`
- Use `memcpy` instead of `memmove` in `__relocate_a_1`
- PR110377 Early VRP and IPA-PROP should work out value ranges from `__builtin_unreachable`

Recognize pattern

```
size_t __len = _M_check_len(...);
if (__len <= 0)
  __builtin_unreachable ();
```

and make GCC realize that length is always positive. Propagate this information across returned values.

PR109849 suboptimal code for vector walking loop

Some commits solving this problem:

- optimize `std::vector::push_back`
- Use `memcpy` instead of `memmove` in `__relocate_a_1`
- PR110377 Early VRP and IPA-PROP should work out value ranges from `__builtin_unreachable`
- Analyze SRA candidates in ipa-fnsummary

It turns out LLVM inlines `_M_realloc_append` due to multiple-accounting bug predicting SRA optimization. I fixed GCC to account it correctly, but that is not enough to get function inlined.

PR109849 suboptimal code for vector walking loop

Some commits solving this problem:

- optimize `std::vector::push_back`
- Use `memcpy` instead of `memmove` in `__relocate_a_1`
- PR110377 Early VRP and IPA-PROP should work out value ranges from `__builtin_unreachable`
- Analyze SRA candidates in ipa-fnsummary
- optimize `std::max` early

Recognize conditional memory load produced by:

```
template< class T >
T& max(T& a, T& b);
```

And turn it into actual max operation.

PR109849 suboptimal code for vector walking loop

Some commits solving this problem:

- optimize `std::vector::push_back`
- Use `memcpy` instead of `memmove` in `__relocate_a_1`
- PR110377 Early VRP and IPA-PROP should work out value ranges from `__builtin_unreachable`
- Analyze SRA candidates in ipa-fnsummary
- optimize `std::max` early
- Fix profile of forwarders produced by cd-dce
- Add cold attribute to throw wrappers and terminate
- Compute ipa-predicates for conditionals involving `__builtin_expect_p`
- Fix handling of `__builtin_expect_with_probability` and improve first-match heuristics

PR109849 suboptimal code for vector walking loop

Some commits solving this problem:

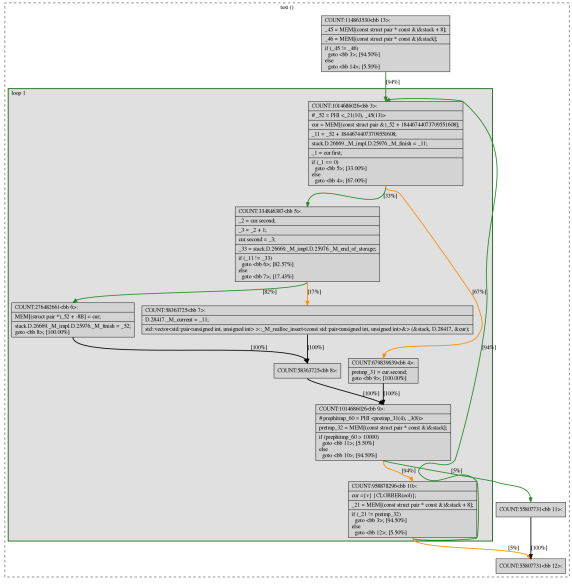
- optimize `std::vector::push_back`
- Use `memcpy` instead of `memmove` in `__relocate_a_1`
- PR110377 Early VRP and IPA-PROP should work out value ranges from `__builtin_unreachable`
- Analyze SRA candidates in ipa-fnsummary
- optimize `std::max` early
- Fix profile of forwarders produced by cd-dce
- Add cold attribute to throw wrappers and terminate
- Compute ipa-predicates for conditionals involving `__builtin_expect_p`
- Fix handling of `__builtin_expect_with_probability` and improve first-match heuristics
- R. Biener: optimize code hoisting

PR109849 suboptimal code for vector walking loop

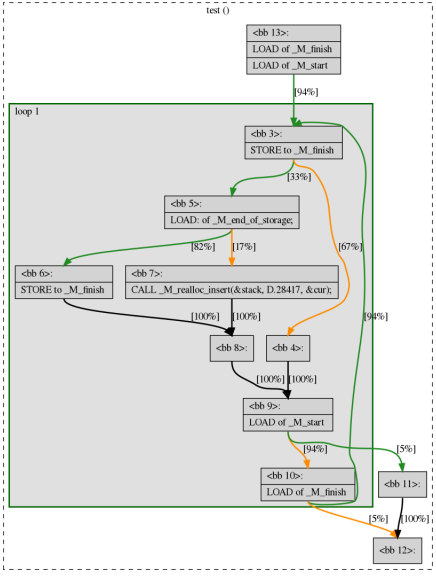
Some commits solving this problem:

- optimize `std::vector::push_back`
- Use `memcpy` instead of `memmove` in `__relocate_a_1`
- PR110377 Early VRP and IPA-PROP should work out value ranges from `__builtin_unreachable`
- Analyze SRA candidates in ipa-fnsummary
- optimize `std::max` early
- Fix profile of forwarders produced by cd-dce
- Add cold attribute to throw wrappers and terminate
- Compute ipa-predicates for conditionals involving `__builtin_expect_p`
- Fix handling of `__builtin_expect_with_probability` and improve first-match heuristics
- R. Biener: optimize code hoisting
- M. Jambor: SRA of non-escaped aggregates passed by reference to calls
- PR110378 IPA-SRA for destructors
  1. M. Jambor: Don't consider CLOBBERS as writes preventing splitting
  2. M. Jambor: Allow IPA-SRA in presence of returns which will be removed
- M. Jambor: Avoid returns of references to SRA candidates

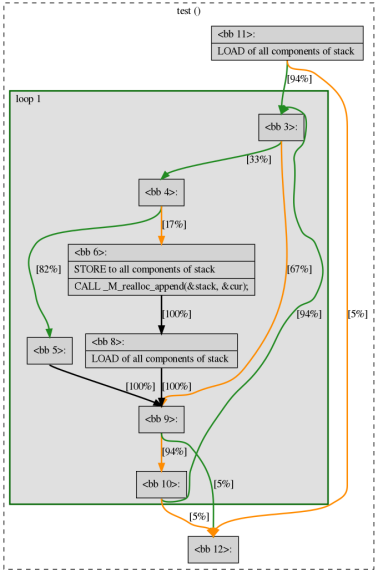
Loop of the simple testcase compiled by GCC13



Loads and stores in the loop of the simple testcase (GCC 13)



Loads and stores in the loop of the simple testcase (GCC 14)



Scalar replacement of Aggregates (SRA)

```
struct Z
{
  int i;
  short j,k;
};
```

```
int foo (void)
{
  struct Z z;
  for (z.i = 0; z.i < bound; z.i++)
    {
      z.j = foo();
      /* ... */
    }
}
```

```
int foo (void)
{
  int z$i;  short z$j;
  for (z$i = 0; z$i < bound; z$i++)
    {
      z$j = foo();
      /* ... */
    }
}
```

Scalar replacement of Aggregates (2)

```
struct Z
{
  int i;
  short j,k;
};
```

```
int foo (void)
{
  struct Z z;
  for (z.i = 0; z.i < bound; z.i++)
    {
      z.j = foo();
      /* ... */
      bar(z);
      /* ... */
    }
}
```

```
int foo (void)
{
  struct Z z;  int z$i;  short z$j;
  for (z$i = 0; z$i < bound; z$i++)
    {
      z$j = foo();
      /* ... */
      z.i = z$i;
      z.j = z$j;
      bar(z);
      /* ... */
    }
}
```

Scalar replacement of Aggregates (3)

```
int foo (void)
{
  struct Z z;
  for (z.i = 0; z.i < bound; z.i++)
    {
      z.j = foo();
      /* ... */
      bar(&z);
      /* ... */
    }
    /* ... */
}
```

Scalar replacement of Aggregates (3)

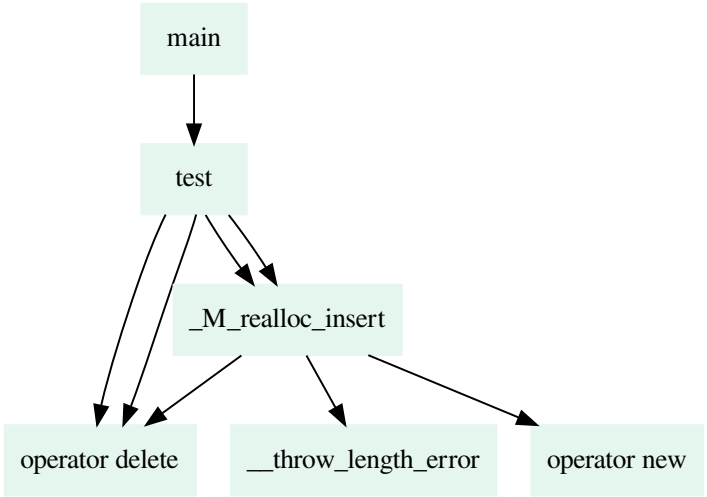
```
int foo (void)
{
  struct Z z;
  for (z.i = 0; z.i < bound; z.i++)
    {
      z.j = foo();
      /* ... */
      bar(&z);
      /* ... */
    }
    /* ... */
}
```

```
int foo (void)
{
  struct Z z;  int z$i; short z$j;
  for (z$i = 0; z$i < bound; z$i++)
    {
      z$j = foo();
      /* ... */
      z.i = z$i;
      z.j = z$j;
      bar(&z);
      z$i = z.i;
      z$j = z.j;
      /* ... */
    }
    /* ... */
}
```

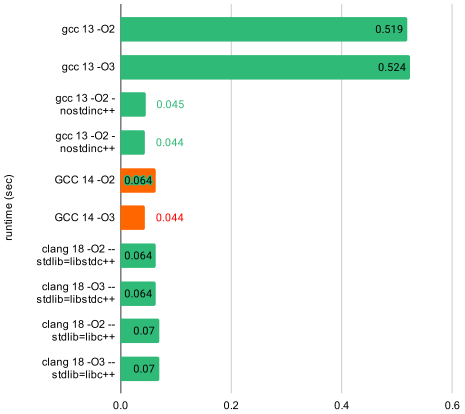
Disqualification of candidates for SRA

```
-  /* Allow constant-pool entries that "need to live in memory".  */
-  if (needs_to_live_in_memory (var) && !constant_decl_p (var))
+
+  if ((is_global_var (var)
+       || (TREE_ADDRESSABLE (var)
+           && pt_solution_includes (&cfun->gimple_df->escaped_return, var))
+       || (TREE_CODE (var) == RESULT_DECL
+           && !DECL_BY_REFERENCE (var)
+           && aggregate_value_p (var, current_function_decl)))
+      /* Allow constant-pool entries that "need to live in memory".  */
+      && !constant_decl_p (var))
```

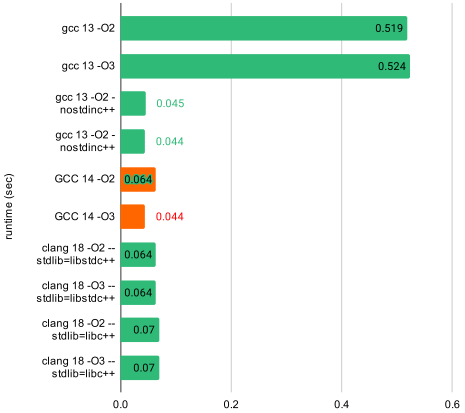
Call graph after interprocedural optimization

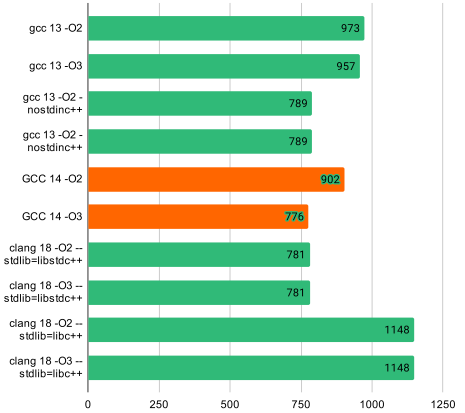


Benchmark runtime and size (GCC vs Clang; libstdc++ vs libc++)



Benchmark runtime and size (GCC vs Clang; libstdc++ vs libc++)





Future work

Work in progress

- PR114821 `_M_realloc_append` should use memcpy instead of loop to copy data when possible
- PR114822 ldist should produce memcpy/memset/memmove histograms based on loop information converted

TODO

- PR110379 Unnecessary copies after early opts (unsolved)
- PR110414 Dead Code Elimination Regression since r14-1127-g9e2017ae6ac (unsolved)
- Adding `-fsane-operator-new` command line option?
- Can we do enought of inter-procedural optimization to figure out that the testcase reallocates the vector just once?
- What about other libstdc++ datastructures?