Portable optimisation hints

Timur Doumler (papers@timur.audio)

Document #: P1774R0
Date: 2019-06-17
Project: Programming Language C++
Audience: Evolution Working Group

Abstract

We propose a standard facility providing the semantics of existing compiler intrinsics such as `__builtin_assume` (Clang) and `__assume` (MSVC, Intel) that tell the compiler to assume a given C++ expression without evaluating it, and to optimise based on this assumption. This is very useful for high-performance and low-latency applications in order to generate both faster and smaller code.

1 Motivation

All major compilers offer built-ins that allow the developer to tell the compiler to assume a given C++ expression, and optimise based on this assumption. They are very useful for high-performance and low-latency applications in order to generate both faster and smaller code. Benefits include more efficient code generation for mathematical operations, better vectorisation of loops, elision of unnecessary branches, function calls, and more. This is existing practice, but it would be much more accessible and easy-to-use if it were a standardised, portable C++ facility.

1.1 History and context

Adding such portable optimisation hints was already proposed once [N4425] and discussed by EWG in 2015 in Lenexa[1]. The paper was rejected. EWG's guidance was that this functionality should be provided within the proposed contract checking facility (CCF), and not as a separate facility.

Unfortunately, four years later we are now in a situation where this functionality is neither part of the CCF, nor available as a separate facility.

In a companion paper [P1773R0] we argue how Contracts have failed to provide portable optimisation hints, and why, even if we were to modify the CCF to allow literal "assume" semantics, as proposed by [P1607R0] and [P1429R2], a lower-level "assume" facility outside of the CCF is still necessary.

The present paper focuses on the actual proposal to standardise this lower-level facility.

[1] https://cplusplus.github.io/EWG/ewg-closed.html#179

1.2 Existing practice

The major compilers offer the following built-ins providing this functionality:

— MSVC and Intel provide `__assume(`*`expression`*`);`

— Clang provides `__builtin_assume(`*`expression`*`);`

— GCC does not provide an analogous built-in directly, but the same can be achieved with `if (!`*`expression`*`) __builtin_unreachable();`

See [N4425] for a more thorough discussion.

1.3 Examples

Consider the following function:

```
int divide_by_32(int x)
{
  __builtin_assume(x >= 0);
  return x/32;
}
```

Without the assumption, the compiler has to generate code that works correctly for all possible input values. With the assumption, it can implement the calculation using a single instruction (shift right by 5 bits). Here is the output generated by clang (trunk) with `-O3`:

Without `__builtin_assume`:

```
mov eax, edi
sar eax, 31
shr eax, 27
add eax, edi
sar eax, 5
ret
```

With `__builtin_assume`:

```
mov eax, edi
shr eax, 5
ret
```

Another example: consider looping over an array of numbers and performing math on the elements. Often, there are invariants on the array size such as: it's a power of two, it's a multiple of the SIMD register size, etc (all very common e.g. in audio processing code). Telling the optimiser about such invariants leads to a much better optimisation and vectorisation of the loop:

```
void limiter(float* buffer, size_t size)
{
  __builtin_assume(size % 8 == 0);
  for (size_t i = 0; i < size; ++i)
     data[i] = std::clamp(data[i], -1.0f, 1.0f);
}
```

For this function, clang (trunk) with `-O3` generates 70 lines of assembly without the assumption, and only 42 lines with it.

See [Regehr2014] for more examples and use cases.

2 Proposed solution

2.1 Semantics

The design goal is to provide a portable facility closely following the compiler built-ins `__assume` and `__builtin_assume`, therefore standardising existing practice. The facility should be implementable

with the existing built-ins on those compiler implementations who have them, without unnecessarily constraining implementations who do not. Therefore, we propose the following semantics:

— It is a statement around a single C++ expression contextually convertible to `bool`.

— The expression is guaranteed to be unevaluated. Therefore, expressions with side effects, such as `++ptr != end`, are allowed and unproblematic.

— However, the behaviour is undefined if the expression would evaluate to `false` (this allows the compiler to optimise the whole program based on the assumption that the expression would always evaluate to `true`).

— Simply ignoring the whole statement is a conforming implementation (even though such an implementation would not give the user the desired code optimisation).

2.2 Syntax

We propose two alternatives how to spell this statement, each with its pros and cons which are discussed below. We leave it up to EWG to give guidance on which should be preferred.

Regardless of which is chosen, we propose that the word "assume" is the word used to spell it. This is the word already used in existing built-ins, and therefore the one that will be least surprising and most self-explanatory to the user. It is also the word that best describes the semantics of this facility.

2.2.1 Alternative 1: Function-style syntax

We could introduce the feature as a "magic" library function, so `__builtin_assume(`*`expression`*`)` becomes:

`std::assume(`*`expression`*`);`

Pros:

— Does not require a core language change.

— Is similar to the closely related `std::assume_aligned` [P1007R3], which was also changed by EWG from an attribute syntax [P0886R0] to a function-style syntax.

— Makes it clear that this facility is not part of Contracts by using a very different syntax.

Cons:

— Would introduce a weird novelty: something that is syntactically a function call, yet does not evaluate its operand.

2.2.2 Alternative 2: Attribute syntax

Alternatively, we could use an attribute syntax, so `__builtin_assume(`*`expression`*`)` instead becomes:

`[[assume:` *`expression`*`]]`

Pros:

— Is similar to other optimisation hint attributes such as `[[likely]]`/`[[unlikely]]` [P0479R5].

— Makes it clear that ignoring this statement does not change the observable semantics of a valid program.

Cons:

— Requires an addition to the core language.

— Too similar to Contracts syntax, even though this feature is not part of Contracts, potentially causing confusion.

2.3 Wording

The formal wording for this proposal will be provided as soon as we receive EWG guidance on the two syntax alternatives.

References

[N4425] Hal Finkel. Generalized Dynamic Assumptions. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4425.pdf, 2015-04-07.

[P0479R5] Clay Trychta. Proposed wording for likely and unlikely attributes (Revision 5). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0479r5.html, 2018-03-16.

[P0886R0] Timur Doumler. The assume aligned attribute. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0886r0.pdf, 2018-02-12.

[P1007R3] Timur Doumler. std::assume_aligned. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0627r3.pdf, 2018-11-07.

[P1429R2] Joshua Berne and John Lakos. Contracts That Work. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1429r2.pdf, 2019-06-14.

[P1607R0] Joshua Berne, Jeff Snyder, and Ryan McDougall. Minimizing Contracts. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1607r0.pdf, 2019-03-10.

[P1773R0] Timur Doumler. Contracts have failed to provide a portable "assume". http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p1773r0.pdf, 2019-06-17.

[Regehr2014] John Regehr. Assertions Are Pessimistic, Assumptions Are Optimistic. https://blog.regehr.org/archives/1096, 2014-02-05.