Document Number: N4425
Date: 2015-04-07
Hal Finkel (hfinkel@anl.gov)

Generalized Dynamic Assumptions

Introduction

The information that a compiler's optimizer, and other static analysis tools, can derive is limited, often in principle by the unknown nature of external functions and processes, and because of fundamental mathematical complexities. Nevertheless, some information potentially-useful to such tools is known by the programmer, and providing a facility to allow the programmer to communicate that knowledge to the implementation has proved important in practice. Such information can, for example, allow a compiler to transform loops (including vectorization on some architectures) and eliminate unneeded comparisons and branches.

Existing practice in this area is well established:

- Microsoft Visual Studio (starting in 2003), as provided the intrinsic __assume(expression), such that the optimizer may assume that the provided expression will evaluate to true at runtime. Accordingly, should the expression evaluate to false, the optimizer may assume the statement is unreachable.
- Intel's compiler provides the generalized __assume intrinsic, in addition to a specialized __assume_aligned intrinsic for pointer alignments. Intel also provides the align_value pseudo-type attribute to provide assumptions on a pointer value's alignment (the attribute does not affect the type, but does attach meaningfully to typedefs).
- IBM's XL compiler does not provide a general assumption facility, but the compiler for the BlueGene systems provides an __alignx intrinsics for providing pointer alignment assumptions, and XL generally provides '#pragma ibm min_iterations' and '#pragma ibm max_iterations' which provide relational constraints on loop induction variables.
- Clang provides the intrinsic __builtin_assume (and __assume in MSVC-compatibility mode) with the same properties as Microsoft's __assume. Clang also provides __builtin_assume_aligned for GCC compatibility, in addition to several other GCC attributes (and Intel's align_value attribute), all of which are implemented using the same internal infrastructure used for __builtin_assume.
- GCC does not explicitly provide a general assumption facility, but general assumptions can be encoded using a combination of control flow and the __builtin_unreachable intrinsic. Also, GCC provides facilities for several important special cases: the __builtin_assume_aligned intrinsic to provide pointer-alignment assumptions, and the nonnull and returns_nonnull attributes for pointer arguments and return values respectively.

At least in the Microsoft, Intel and Clang implementations, it is important to note that the expression provided to the assumption intrinsic is not actually evaluated. Any side effects of the expression are discarded (a warning about discarded side effects is issued instead). So:

```
__assume(++i); // This does not actually increment i.
```

Proposal Overview

The existing implementations that provide generic assumptions use some keyword in the implementation-reserved identifier space (__assume, __builtin_assume, etc.). Because the expression argument is not evaluated (side effects are discarded), specifying this in terms of a special library function (e.g. std::assume) seems difficult. A new keyword (e.g. assume) could be introduced, but that can also be avoided by reusing existing keywords. Specifically in this proposal, reusing the true and false keywords (where false has been included to maintain symmetry). A small enhancement to alignof is proposed to allow this assumption facility to more-naturally handle pointer alignments.

```
true(i == 5); // The compiler may assume that, at runtime, i will
              // always equal 5 when evaluating expressions
              // appearing after this.

true(false);  // This statement must be unreachable.

false(i < 2); // i is not less than 2 here.

void foo(int i) {
  true(i > 6);  // i is always greater than 6 here.
  if (i < 3)    // This condition can be statically evaluated.
    bar();
  …
 }

 void bar(float *q, const float *p, int n, int m) {
   true(alignof(p) == 16 && alignof(q) == 16);
   true(m % 16 == 0);

   // This loop can safely be interleaved (modulo unrolled)
   // and/or vectorized by some factor no greater than 16,
   // based on the information provided, without runtime checks.
   for (int i = 0; i < n; ++i)
     q[i] = p[i] + p[i+m];
 }
```

```
extern int foo();
true(foo() >= 42); // The expression here is unevaluated,
                   // so no function call will be generated.
```

Relationship to Prior Proposals

This proposal is not based on any previous proposal. There is some conceptual similarity between this proposed facility, and assert (along with proposed enhancements for optionally-checked contracts, such as N4293). assert and related facilities, however, are specifically designed to enhance program robustness and aid debugging. This facility, on the other hand, is intended to aid the compiler's optimizer, and so its use could decrease program robustness in exchange for better performance.

Implementation Concerns

Many implementations (Microsoft, Intel, Clang, etc.) already support this feature, albeit with a slightly different syntax. Because an implementation is not required to make use of the information provided by this facility, the basic implementation cost should be very low.

Proposed Changes

In 5p8, add the relevant subclause(s) added to Clause 5 here to the list of contexts in which unevaluated operands appear.

Add to 5.3p1 a new expression form to the list of unary-expression with the syntax:

assumption-expression:
`true` (*expression*)
`false` (*expression*)

and also add to the list of unary-expression:

`alignof` *unary-expression*

Add to 5.3 a new subclause, 5.X, "Assumptions" with the text:

p1: The `true` and `false` operators do not evaluate their operand, however, if the operand were to be evaluated, contextually converted to `bool`, and the result of that conversion would not be `true` for the `true` operator, or `false` for the `false` operator, the behavior of the program is undefined. If the contextual conversion to `bool` is ill-formed, then the program is ill-formed.

p2: The result of the operator is a constant of type `bool`, `true` for the `true` operator and `false` for the `false` operator.

In 5.3.6, replace the first paragraph with:

p1: `alignof` applied to an expression shall appear only as part of the (unevaluated) operand of a `true` or `false` operator (5.X). The result shall represent a lower bound on the alignment of a pointer to its operand (as though the unary & operator had been applied to generate a pointer).

p2: When `alignof` is applied to a *`type-id`*, the *`type-id`* shall represent a complete object type, or an array thereof, or a reference to one of those types. Such an `alignof` expression yields the alignment requirement of its operand type.

Renumber remaining paragraphs in 5.3.6.