_Dependent_ptr to simplify carries a dependency

Authors: Akshat Garg, Paul E. McKenney, Ramana Radhakrishnan, and TBD
Other contributors: TBD

Abstract

The C standard currently specifies that that `memory_order_consume` loads feed into *carries a dependency,* however, no known implementation does anything other than promote `memory_order_consume` to `memory_order_acquire`. C users therefore avoid `memory_order_consume` loads in favor of volatile loads, inline assembly, and other subterfuge. There has been considerable work within the C++ standards committee to address similar issues in C++, however, the current proposals involve C++ templates, which have no reasonable C equivalent.

This paper therefore proposes a new `_Dependent_ptr` type qualifier to provide this functionality in the C language. It also reports on a prototype implementation of this proposal in GCC.

Introduction

The problems with `memory_order_consume` and the related *carries a dependency* memory-model relation are well known, and have been the subject of lengthy and spirited discussions in WG21:

1. WG21 N4036 re-opened the discussion of `memory_order_consume`, which in all known implementations is a synonym for `memory_order_acquire`. It presents usage within the Linux kernel, including a survey of Linux-kernel use cases. It also describes the high-performance alternatives to `memory_order_consume` that are used within the Linux kernel along with the restrictions on the use of these alternatives. It closes with lists of alternative approaches to providing a high-performance `memory_order_consume` in real C and C++ implementations.
2. WG21 N4215, WG21 N4321, and WG21 P0098R0 (later updated by P0098R1) refine and expand on the aforementioned WG21 N4036. One key refinement was the exclusion of integers from the set of types permitted to carry dependencies, and patches were accepted into the Linux kernel removing the integer use case. (Dependency-carrying integral types have since crept back in, but will be dealt with one way or another.)
3. WG21 P0190R0 (later updated by P0190R1, P0190R2, P0190R3, and P0190R4) took the approach of formalizing the Linux-kernel usage restrictions presented in the earlier papers (and documented officially by Linux-kernel rcu_dereference.txt). This approach was discussed at length and eventually rejected due to the burden on both implementers and users.
4. WG21 P0462R0 contained a number of alternative proposals for marking variables intended to carry dependencies. A later update (P0462R1) chose a C++ template approach.
5. WG21 P0750R0 (later updated by P0750R1) used a separated dependency object maintained by inline assembly language. This approach permitted a very simple implementation and was very easy to use, but had the effect of doubling the size of pointers in simple implementations. This expansion in size could in theory be optimized away, but in practice this would require substantial changes.

This proposal returns to the approach proposed by WG21 P0462R1, but using a C-language type qualifier instead of the C++ template-based approach. This new `_Dependent_ptr` type qualifier is used to explicitly mark variables that are intended to carry a dependency so that dependency-breaking optimizations need be disabled only for those variables

and any temporaries created to hold values computed from marked variables. Note that function parameters and return values that are intended to carry dependencies must also be marked with `_Dependent_ptr`.

Syntax

A dependency-carrying pointer is declared as follows:

```
int * _Dependent_ptr p;
```

A function that takes a dependency-carrying pointer as a parameter and also returns one is declared as follows:

```
int * _Dependent_ptr foo(int * _Dependent_ptr bar);
```

Note that the `_Dependent_ptr` will normally go between the asterisk and the name above. The following two examples contrast placement before and after the asterisk:

1. T * _Dependent_ptr x;
   [This implies that variable x is dependent pointer qualified. It is the usual way of declaring.]

2. T _Dependent_ptr * x;
   _Dependent_ptr T * x;
   [Valid only when T is a pointer type, so if there is a U* equivalent to T, then we have “U * _Dependent_ptr *x”.]

   Consider the example below:
   T* _Dependent_ptr *y;
   typedef T* U;
   Then we get effectively “U _Dependent_ptr *y;” which gets us back to case 1.

Future note: The _Dependent_ptr type qualifier may someday support dependency ordering through array indexes and perhaps also dependency ordering through integral types converted to a pointer and vice-versa.

Implementations might also choose to provide command-line arguments or similar to permit compatibility with code bases (such as the Linux kernel) that do not mark dependency-carrying pointers.

Interaction With Other Traits

The `_Dependent_ptr` type trait can be combined with other traits:

1. `_Alignas`: Dependency-breaking optimizations would be avoided, and the variable would be aligned as specified.
2. `_Atomic`: Dependency-breaking optimizations would be avoided, and the variable would be accessed using C11 atomics.
3. `const`: This is not particularly useful for variables with static storage duration because compile-time initialization does not require dependency ordering, but then again, use of `_Dependent_ptr` on such variables is suspect to

begin with. Otherwise, the `const _Dependent_ptr` variable would normally be initialized from another `_Dependent_ptr` variable or from a `memory_order_consume` load. The variable would disallow further stores and avoid breaking dependencies.

4. `extern`: Dependency-breaking optimizations would be avoided, and the variable would be usable within other translation units. This is also an unusual addition to a `_Dependent_ptr` unless also accompanied by `_Thread_local` because there are no known non-suspect multi-threaded-access use cases for `_Dependent_ptr`.
5. `register`: Dependency-breaking optimizations would be avoided, and the compiler would be given a hint to keep the variable in a register.
6. `restrict`: Dependency-breaking optimizations would be avoided, and the compiler may assume that the pointed-to object is only accessed through this pointer and through pointers derived from it.
7. `static`: Dependency-breaking optimizations would be avoided, and the variable would be static. This is also an unusual addition to a `_Dependent_ptr` unless also accompanied by `_Thread_local` because there are no known non-suspect multi-threaded-access use cases for `_Dependent_ptr`.
8. `_Thread_local`: The dependency-carrying variable is thread-local, and avoids dependency-breaking optimizations.
9. `volatile`: All accesses would be executed as per the abstract machine, and dependency-breaking optimizations would be avoided.

Conversions

The following subsections describe conversions adding and discarding _Dependent_ptr qualifiers, conversions from `memory_order_consume` loads, conversions adding and discarding other qualifiers, and finally conversions involving pointer arithmetic and temporaries.

Conversion Adding and Discarding `_Dependent_ptr` Qualifiers

There are four cases to consider: [Do we need to define conversions to non pointer types also?]

First, from `T *` to `T * _Dependent_ptr`: Allowed. This assignment marks the start of a new dependency chain. Consider the following example:

```
T *x;
T * _Dependent_ptr y;
y = x;
```

The above can create data race if `x` can change. Therefore, there are two use cases for this type of assignment:

```
y = rcu_dereference(x); // Linux kernel
y = atomic_load_explicit(&x, memory_order_consume); // C11
```

Assigning from an unqualified to a `_Dependent_ptr` qualified pointer prevents future dependency breaking for `y`, but does nothing to prevent prior dependency breaking for `x`. This has implications covered in a later section.

Second, from `T *` to `T _Dependent_ptr *`: Allowed only when `T` is pointer type. Consider the example:

```
T *x;
T _Dependent_ptr *y;
```

```
y = x;
```

Note that the `_Dependent_ptr` is not involved in the assignment, but we have a pointer to another pointer that preserves dependency ordering. Some implementations may choose to emit a warning in this case, similar to the situation with the `const` keyword. Such implementations should of course suppress their warnings when an explicit cast or a `kill_dependency()` is involved. Use of casts or `kill_dependency()` is appropriate in cases where the pointer has come under the protection of a lock or reference count. See for example use of `rcu_pointer_handoff()` within the Linux kernel.

Third, from `T * _Dependent_ptr` to `T *`: Allowed. Consider the example:

```
T *x;
T * _Dependent_ptr y;
x = y;
```

Although the dependency chain continues through `y`, `x` does not carry a dependency.

Fourth and finally, from `T _Dependent_ptr *` to `T *`: Allowed only when `T` is pointer type. Consider the example:

```
T *x;
T _Dependent_ptr *y;
x = y;
```

Note that the `_Dependent_ptr` is not involved in the assignment, but we have a pointer to another pointer that preserves dependency ordering regardless of the variable that the pointer is accessed through.

Conversions From `memory_order_consume` Loads

Again, note that converting from an unqualified pointer to a pointer qualified with `_Dependent_ptr` does nothing to prevent dependency-breaking optimizations from being applied to the unqualified pointer. This means that a `memory_order_consume` load must return a pointer qualified with `_Dependent_ptr`. To see this, note that if such loads returned unqualified pointers, dependency-breaking optimizations could be applied at the time of the `memory_order_consume` load.

Of course, the user might write "`p = atomic_load_explicit(&x, mo)`", where the value of `mo` is unknown at compile time. In this case, a high-QoI implementation could look at the declaration of `p`, and make the type of `atomic_load_explicit()` be qualified by `_Dependent_ptr` only if `p` is `_Dependent_ptr`. Similarly, if the user wrote "`return atomic_load_explicit(&x, mo)`", a high-QoI implementation could check to see if the function's return type was qualified with `_Dependent_ptr`. to determine whether the type of `atomic_load_explicit()` should be qualified by `_Dependent_ptr`.

A low-QoI implementation could simply act as if the return type of `atomic_load_explicit()` way always qualified by `_Dependent_ptr`.

Conversions Adding and Discarding Other Qualifiers

The class of conversions adding and discarding other qualifiers from variables qualified with `_Dependent_ptr` operates as expected, but is spelled out for completeness.

1. `_Atomic`: Default C-language access to these variables is sequentially consistent (`memory_order_seq_cst`), which guarantees strong ordering and high overhead. In all known use cases, the strong ordering obviates any `_Dependent_ptr` qualification.
2. `const`: The compiler's assumption of and enforcement of lack of updates changes as one would expect from similar conversions involving unqualified variables.
3. `restrict`: The compiler's aliasing assumptions change as one would expect from similar conversions involving unqualified variables
4. `volatile`: The compiler's observance of optimization restrictions inherent to `volatile` changes as one would expect from similar conversions involving unqualified variables.

Implementations are expected to use warning strategies similar to those in place for variables not qualified with `_Dependent_ptr`.

Arithmetic and Temporaries

Any expression involving a `_Dependent_ptr` variable should generate a `_Dependent_ptr` qualified result.

Any temporary variables generated for `_Dependent_ptr` qualified expressions should also be `_Dependent_ptr` qualified.

GCC Implementation

Currently, we have a prototype implementation for `_Dependent_ptr` type qualifier upto RTL phase. This qualifier is allowed for pointer types only. There are four phases in implementation:

1. **Front-end:** The front-end parses the `_Dependent_ptr`. The patch is here. In this phase, we also check that this qualifier gets applied to pointer types only. Some test cases have been added from the document P0190R4.
2. **TREE phase:** Many optimizations can be problematic like constant propagation, etc during this phase. Therefore, we are considering the `_Dependent_ptr` qualifier as volatile during this phase. The patch is here.
3. **RTL phase:** Some optimizations can be problematic like common sub-expression elimination, instruction combination, dead store elimination, etc. We have again considered `_Dependent_ptr` as volatile here. Also, we consider this for only `MEM` type expressions or RTX_OBJ. In any RTL dump, `MEM` with `'/d'` denotes that the expression is containing some `_Dependent_ptr` qualified expression. The patch is here.
4. **Target specific:** We are currently targeting only one or two weakly-ordered CPUs. Our focus is presently on `POWER9` architecture. We will update the details as we proceed in this phase.

The implementation also issues the following warnings:

1. Function pointer: Types are not supported currently. Error message for when declared under this qualifier:

`error: ‘_Dependent_ptr’ requires a pointer (‘function pointer’ is invalid)`

2. Non-pointer types: Currently dependencies are maintained for non-pointer types, like int, float, etc. Using with any of them generates the following error message:
   `error: ‘_Dependent_ptr’ requires a pointer (‘float’ is not supported)`
3. Arrays: Currently, this qualifier cannot be used with array declarations. Using with arrays can lead to following error message;
   `error: ‘_Dependent_ptr’ requires a pointer (arrays are not supported currently)`

Summary

This paper proposes a C-language `_Dependent_ptr` type qualifier to mark variables that the user intends to carry dependencies. As the name suggests, this type qualifier is intended to be used only on pointers, though we request feedback on future possibilities such as applying it to array indexes as well. We also welcome feedback on the prototype GCC implementation and on how this type qualifier would fit into other C implementations.