Document number: P1465R0
Date: 2018-01-20 (pre-Kona)
Reply-to: David Goldblatt <davidtgoldblatt@gmail.com>
Audience: EWG[I], SG14 (if interested)

P1465R0: Function optimization hint attributes: [[always_inline]], [[never_inline]]

Audience: EWG[I], SG14 if interested

Introduction

Implementation-specific hints to the optimizer telling it that a given function should or shouldn't be inlined into callers whenever possible are widely supported and used. We propose standardizing the names for them.

This paper suggests some specific choices for those names, and some wording (excluding the details specific to the grammar, which are outside my area of expertise). These are placeholders that can be bikeshed.

The goal

Inline the indicated function whenever possible, or stop it from being inlined wherever possible.

When is it useful?

These come in handy when the compiler makes bad inlining decisions on its own. There can be a variety of causes (not all of them unreasonable, or indicative of a flaw in the compiler). Here are some possible situation where a function that ought to be inlined (in some fuzzy, application-specific sense) won't be:

- The function looks big, but will shrink after constant propagation (of an constant argument at the call site) and dead code elimination.
- The function is in a translation unit compiled such that it is optimized for space (e.g. -Os); only few a few very hot functions should be inlined into to their call sites.
- The function has some inline assembly with lots of newlines in it. (Not a joke).

[[never_inline]] can also be beneficial to performance; unnecessary inlining of slow paths can waste cache and make other optimization passes less effective.

In general, programmers can be guided by benchmarking data in order to determine which functions should have one of the attributes applied; they don't necessarily need to understand or reason about all the internals of a modern optimizing compiler to benefit.

There are also times when fine-grained control of inlining decisions is important for debuggability. Consider a profiling library, which may have code like the following:

```
// In SomeProfilingLibrary.hpp
inline void profileEvent(ProfieToken& token) {
  if (--token.counter < 0) {
    doSlowPath(token);
  }
}
// In SomeProfilingLibrary.cpp
void doSlowPath(ProfileToken& token) {
  token.resetCounter();
  auto stackTrace = grabStackTrace();
  // The number of library-internal frames to exclude from the stack;
  // the user shouldn't see profileEvent() or doSlowPath() in profiles.
  int kProfilingLibFrames = ???;
  recordStackTrace(stackTrace.begin() + kProfilingLibFrames,
      stackTrace.end());
}
```

Here, to give accurate stack traces, we want to make kProfilingLibFrames a constant. If profileEvent is inlined into some call sites but not others, we'll skip an incorrect number of frames. We can therefore apply [[always_inline]] to profileEvent, and [[never_inline]] to doSlowPath. No such strategy will be completely portable (e.g. can the stack unwinder read debug info or not? If it can, it might reconstruct the inlined frame), but this one will typically ensure that at least the value is constant within a given build mode on a given implementation (at which point the correct value could even be detected at runtime if desired).

Issues

The naming leaves something to be desired; it's not necessarily possible to ensure that an [[always_inline]] function is always inlined (suppose it's recursive, or its body is not available, or the calling function is running up against some implementation limit). However, existing implementations use "always" or "force", and less strong names don't convey the same level of

intensity. [[should_inline]] and [[should_not_inline]], [[try_inline]] and [[try_no_inline]], and [[pretty_please_inline]] and [[pretty_please_noinline]] are all possible alternatives.

Should [[always_inline]] require the inline specifier? Should [[never_inline]] disallow it?

No. The inline specifier is an ODR escape hatch, and [[always_inline]] and [[never_inline]] are optimization hints. [[always_inline]] can make sense on non-inline functions (e.g. causing inlining in static functions, or during LTO), and [[never_inline]] can make sense on inline ones (indeed, this is handy in debugging optimized builds; applying [[never_inline]] to an inline function in a header is often the simplest way to ensure that the function appears in a stack trace, or can have a single tracepoint inserted on it).

Straw-person wording

I suggest

X.Y.Z.1 The attribute-tokens always_inline and never_inline shall appear at most once in each … [I don't feel equipped to complete this paragraph].

X.Y.Z.2 [Note: The use of the always_inline attribute is intended to indicate to the implementation that the body of the function declared always_inline should be included into the body of calling functions whenever feasible. The use of the never_inline attribute is intended to indicate to the implementation that the body of the function declared never_inline should not be included into the body of the calling function whenever feasible)]