Are modules fast?

Rene Rivera – grafikrobot@gmail.com – PxxxxR0, 2019-01-21

Table of Contents

Document number	ISO/IEC/JTC1/SC22/WG21/PxxxxR0
Date	2019-01-21
Reply-to	Rene Rivera, grafikrobot@gmail.com
Audience	WG21

1. Abstract

Measurements of the performance of building C++ modules and relevant comparisons.

2. Changes

2.1. R0 (Initial)

Initial performance measurements for synthetic tests of gcc merged modules implementation (rev 268043, 2019-01-17 10:58:47 -0600 (Thu, 17 Jan 2019)) .

3. Introduction

One of the stated goals of the modules proposals was performance over existing non-modular source specifically in terms of scalable builds. This paper aims to answer the important question of wether that goal was achieved by current modules implementations.

Building software does not happen in the solitude of the C++ compiler. It is a careful orchestration of a collection of tools from the preprocessor, compiler, linker, assembler, etc controlled by the build system. These have all been optimized to deal with the current separate compilation model and generally perform gargantuan feats. But as such it means that performance measurement needs to take into account, to some degree, all those in a controlled method to generate meaningful measurements.

To that end the measurements presented here are "synthetic". They are structured such that they simulate and isolate certain components, like the build system, to facilitate the extraction of relevance in the data. Such methods are described below for individual measurements.

4. Parallel Build

Although there are various aspects comprising performance scalability of key interest is in seeing how the potential algorithmic savings of modules "caching" compiler work at the cost of longer DAG build chains compares against the current, almost unlimited, parallelized building of plain source compiles. This test aims to measure the overall compile times of modular source across varied dependency chain depth against equivalent non-modular source.

4.1. Method

Overall we perform the basic task of compiling 300 C++ source files into corresponding object files only. We perform the compile at different dependency DAG chain depth counts. Each source depends, either as an `import` or `#include`, on some number of sources (headers or modules) from all the previous DAG levels. The overall time to compile all 300 files is taken as the result of the measurements.

4.1.1. Modular Source

For modular source this test simulates the behavior of a build system to compile individual modular source files in parallel executions of the compiler to the limit of the available CPU threads. It does so in appropriate, but simulated, dependency DAG order; compiling each DAG level as a group. The process goes as such:

1) Generate a synthetic, and simplified, DAG to describe the build. Where all the sources in the level can be compiled in parallel.

2) Generate all source files similar to this:

```C++
export module m148;

import m0;
import m15;
import m84;

namespace m148_ns
{
export int n = 0;
export int i1 = 1;
// ...
export int i300 = 300;
}
```

Where the imports are randomly chosen from the set of all already compiled modules (in previous DAG levels).

3) For each DAG level compile all the source files therein with a GCC invocation similar to:

```BASH
g++ -fmodules-ts -c -O0 m148.cpp
```

4.1.2. Non-modular Source

For no-modular source this test simulates the behavior of a build system to compile individual all source files in parallel executions of the compiler to the limit of the available CPU threads. As the non-modular sources only depend on already existing header source files all source files can be attempted to be compiled at once. The process goes as such:

1) Generate a synthetic, and simplified, DAG to describe the build with parity to the modular build. But which is not used in the build itself.

2) Generate all source header files similar to this:

```C++
#ifndef H_GUARD_h148
#define H_GUARD_h148
#include "h77.hpp"
#include "h78.hpp"
#include "h92.hpp"


namespace h148_ns
{
int n = 0;
 int i1 = 1;
// ...
 int i300 = 300;
}
#endif
```

Where the includes are randomly chosen from the set of all headers files in previous DAG levels. Although not needed to limit to previous DAG levels this is done to keep parity with the modular source in terms of statistical complexity of the included source size.

3) Generate all source files similar to this:

```C++
#include "h148.hpp"
```

4) Compile all source files as one group with a GCC invocation similar to:

```BASH
g++ -c -O0 h148.cpp
```

4.2. Limitations

This test has some real-life limitations borne out of the software and hardware used for testing. Some of these limitations where discovered through experimentation, for example with internal compiler errors.

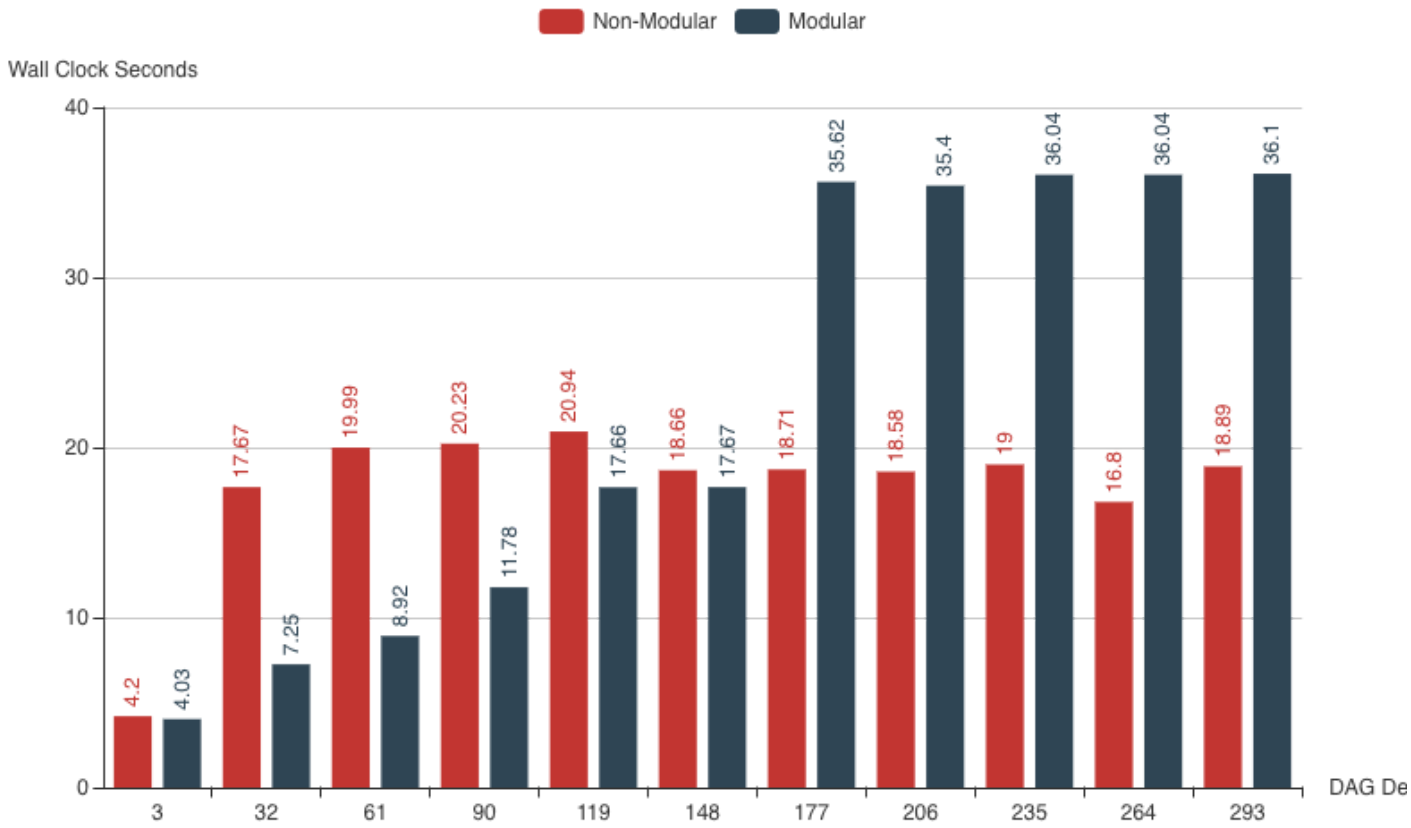
The current test hardware is an OSX laptop with an Intel 4 core and 8 thread CPU. And as such the execution pool for compilation only attempts to execute 8 processes at once.

Being an experimental compiler the support for compiling modules is fragile and placed severe limits on what the modular sources could contain. Most current C++ construct fail to compile reliably, for example templates, and cause ICEs. This is why the generated sources only contain `int` variable definitions.

Along that same line the current modular compiler also seems to have stability limits on how many imports can be done. This is why only a maximum of three (3) `import` statements are included in the source.

This is a simulation with "perfect" build knowledge as all dependency and source information is known before building starts. Hence it is only a very rough approximation of reality where build system have to deal with dependency discovery while building and in the face of generated source files.

4.3. Results



As we can see there is an initial performance advantage to modular source over non-modular source. This advantage diminishes as the dependency depth grows and hence the number of parallel modular source compiles decreases. The performance of the non-modular stays about the same throughout as the amount of header information per source doesn't change substantially. Whereas the modular build reaches a point where the number of parallel builds are either one (1) or two (2) executions making the execution essentially linear.

This indicates that as long as we can execute parallel modular compiles at the same rate as non-modular compiles it appears that modular compiles have a varied but perceptible performance advantage.

This also indicates would indicate that the larger the number of real parallel builds possible the less advantage modular compiles have over non-modular compiles. This would be something to investigate further with more capable hardware.

5. Conclusion

With the limitations of the sampling currently possible the best we can conclude is that there are some possible gains in modular compilation but that the gains might not be attainable for really large and complex source code projects.

6. Acknowledgements

Thanks to Nathan Sidwell for the work to implement the latest merged modules proposal in GCC. And the citizens of the Internet and more specifically the contributors to StackOverflow for the hints that made it possible to decipher how to get the GCC svn checkout build in OSX.

7. References

GCC cxx-modules implementation, Nathan Sidwell https://gcc.gnu.org/wiki/cxx-modules (https://gcc.gnu.org/wiki/cxx-modules)

C++ Tooling Stats, Rene Rivera https://github.com/bfgroup/cpp_tooling_stats (https://github.com/bfgroup/cpp_tooling_stats)