ctbench - compile-time benchmarking and analysis

Jules Penuchot [iD] [1] **and Joel Falcou** [iD] [1]

1 Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, 91400, Orsay, France

Summary

With libraries like Eigen (Guennebaud et al., 2010), Blaze (Iglberger, 2012), and CTRE (Dusíková, 2018) being developed with a large tempalte metaprogrammed implementation, we're seeing increasing computing needs at compile time. These needs might grow even larger as C++ embeds more features over time to support and extend these kinds of practices, like compile-time containers (Dimov et al., 2019) and static reflection (Vandevoorde et al., 2022). However, there is still no clear cut methodology to compare the performance impact of different metaprogramming strategies. And as new C++ features allows for new techniques with claimed better compile-time performance, no proper methodologies are provided to back up those claims.

This paper introduces **ctbench**, a set of tools for compile-time benchmarking and analysis in C++. It aims to provide developer-friendly tools to declare and run benchmarks, then aggregate, filter out, and plot the data to analyze it. As such, **ctbench** is meant to become the first layer of a proper scientific methodology for analyzing compile-time program behavior.

We'll first have a look at current tools for compile-time profiling and benchmarking and establish the limits of current tooling, then we'll explain what **ctbench** brings to overcome these limits.

Statement of need

C++ template metaprogramming raised interest for allowing computing libraries to offer great performance with a very high level of abstraction. As a tradeoff for interpreting representations of calculations at runtime, they are represented at compile time, and transformed directly into their own programs.

As metaprogramming became easier with C++11 and C++17, it became more mainstream and consequently, developers have to bear with longer compilation times without being able to explain them. Therefore, being able to measure compilation times is increasingly important, as is being able to explain them as well. A first generation of tools aims to tackle this issue with their own specific methodologies:

- Buildbench (Tingaud, 2017) measures compiler execution times for basic A-B compile-time comparisons in a web browser,
- Metabench (Dionne et al., 2017) instantiates variably sized benchmarks using embedded Ruby (ERB) templating and plots compiler execution time, allowing scaling analyses of metaprograms,
- Templight (Porkoláb et al., 2009) adds Clang template instantiation inspection capabilities with debugging and profiling tools.

Additionally, Clang has a built-in profiler (Afanasyev, 2019) that provides in-depth time measurements of various compilation steps, which can be enabled by passing the `-ftime-trace`

flag. Its output contains data that can be directly linked to symbols in the source code, making it easier to study the impact of specific symbols on various stages of compilation. The output format is a JSON file meant to be compatible with Chrome's flame graph visualizer, which contains a series of time events with optional metadata like the mangled C++ symbol or the file related to an event. The profiling data can then be visualized using tools such as Google's Perfetto UI.
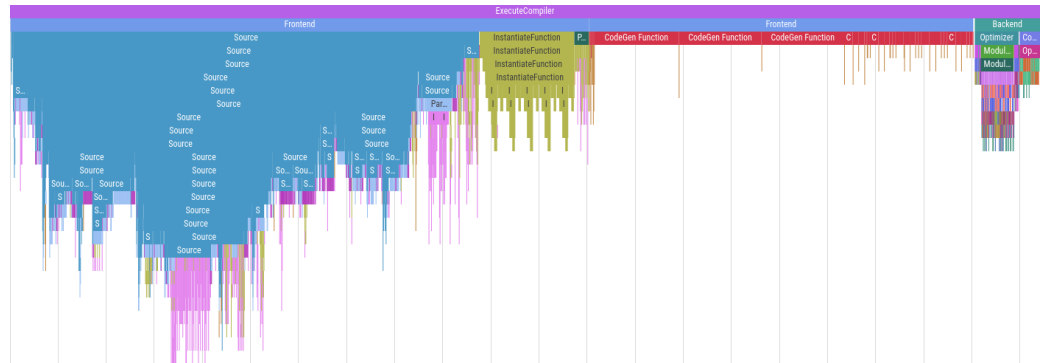

Figure 1: Perfetto UI displaying a sample Clang time trace file

Clang's profiler data is very exhaustive and insightful; however, there is no tooling to make sense of it in the context of variable size compile-time benchmarks. **ctbench** tries to bridge the gap by providing a tool to analyze this valuable data. It also improves upon existing tools by providing a solution that's easy to integrate into existing CMake projects, and generates graphs in various formats that are trivially embeddable in documents like research papers, web pages, and documentation. Additionally, relying on persistent configuration, benchmark declaration and description files provide strong guarantees for benchmark reproductibility, as opposed to web tools or interactive profilers.

Functionality

Originally inspired by Metabench (Dionne et al., 2017), **ctbench** development was driven by the need for a similar tool observes Clang's time-trace files to help get a more comprehensive view on the impact of metaprogramming techniques on compile times. A strong emphasis was put on developer friendliness, project integration, and component reusability.

ctbench provides:

- a well documented CMake API for benchmark declaration, which can be generated using the C++ pre-processor,
- a set of JSON-configurable plotters with customizable data aggregation features and boilerplate code for data handling, which can be reused as a C++ library.

In addition to **ctbench**'s time-trace handling, it has a compatibility mode for compilers that do not support it like GCC. This mode works by measuring compiler execution time just like Metabench (Dionne et al., 2017) and generating a time-trace file that contains compiler execution time. Moreover, the tooling allows setting different compilers per target within a same CMake build, allowing black-box compiler performance comparisons between GCC and Clang for example or comparisons between different versions of a compiler.

All these features make **ctbench** a very complete toolkit for compile-time benchmarking, making comprehensive benchmark quick and easy, and the only compile-time benchmarking tool that can use Clang profiling data for metaprogram scaling analysis.

Statement of interest

ctbench was first presented at the CPPP 2021 conference (Penuchot, 2021), which is the main C++ technical conference in France. It is being used to benchmark examples from the poacher (Penuchot, 2020) project, which was briefly presented at the Meeting C++ 2022 (Paul Keir, 2022) technical conference.

Related projects

- Poacher: Experimental constexpr parsing and code generation for the integration of arbitrary syntax DSL in C++20
- Rule of Cheese: A collection of compile-time microbenchmarks to help set better C++ metaprogramming guidelines to improve compile-time performance

Acknowledgements

References

Afanasyev, A. (2019). *Adds '-ftime-trace' option to clang that produces chrome 'chrome://tracing' compatible JSON profiling output dumps*. https://reviews.llvm.org/D58675

Dimov, P., Dionne, L., Ranns, N., Smith, R., & Vandevoorde, D. (2019). *More constexpr containers*. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0784r7.html

Dionne, L., Dutra, B., Holmes, O., & others. (2017). *Metabench: A simple framework for compile-time microbenchmarks*. https://github.com/ldionne/metabench/

Dusíková, H. (2018). *Compile time regular expression in C++*. https://github.com/hanickadot/compile-time-regular-expressions

Guennebaud, G., Jacob, B., & others. (2010). *Eigen. 3*. https://eigen.tuxfamily.org

Iglberger, K. (2012). *Blaze C++ linear algebra library*. https://bitbucket.org/blaze-lib

Paul Keir, J. P., Joel Falcou. (2022). *Meeting C++ - a totally constexpr standard library*. https://www.youtube.com/watch?v=ekFPm7e___vI

Penuchot, J. (2020). *Poacher: C++ compile-time compiling experiments*. https://github.com/jpenuchot/poacher/

Penuchot, J. (2021). *Ctbench: Compile time benchmarking for clang*. https://www.youtube.com/watch?v=1RZY6skM0Rc

Porkoláb, Z., Mihalicza, J., & Pataki, N. (2009). *Measuring compilation time of C++ template metaprograms*. http://aszt.inf.elte.hu/~gsd/s/cikkek/abel/comptime.pdf

Tingaud, F. (2017). *Build-bench*. https://build-bench.com/

Vandevoorde, D., Childers, W., Sutton, A., & Vali, F. (2022). *P1240R2: Scalable reflection*. https://wg21.link/p1240r2; WG21.