ctbench: compile time benchmarking for Clang

Jules Pénuchot

CPPP 2021 PARIS

MUREX

CODE RECKONS

Science to the CORE

ctbench
compile time benchmarking for Clang

Jules Pénuchot | he/him
PhD Student
Parallel Systems, LISN, Paris-Saclay University

ctbench
compile time benchmarking for Clang

ctbench
set of tools for compile time benchmarking and analysis for Clang

ctbench
compile time benchmarking for Clang

Metaprogramming is evolving

- Support libraries
  - Boost.Mpl,
  - Boost.Fusion,
  - Boost.Hana,
  - Boost.Mp11,
  - Brigand,
  - *that code snippet repository you probably own*

→ People want and build better metaprogramming libraries as time goes

ctbench
compile time benchmarking for Clang

Metaprogramming is evolving

- C++ itself
  - Type traits (C++11)
  - constexpr specifier (C++11)
  - if constexpr (C++17)
  - More constexpr containers (C++20) - https://wg21.link/p0784
  - Reflection proposal - https://wg21.link/p1974
  - Metaprogramming proposal - https://wg21.link/p2237
  - ~~*C++ will become self-aware and take over the Metaverse*~~

→ C++ might finally get a proper **metaprogramming API**

ctbench
compile time benchmarking for Clang

Metaprogramming is evolving

- Applications
  - Eigen - https://eigen.tuxfamily.org/
  - Blaze - https://bitbucket.org/blaze-lib/blaze
  - CTRE - https://github.com/hanickadot/compile-time-regular-expressions
  - CTPG - https://github.com/peter-winter/ctpg

→ Going towards... Compile time compilers?

ctbench
compile time benchmarking for Clang

Is it a good thing? Sure!

- Better **embedded domain-specific languages**
  → Makes C++ libraries easier to interact with for non C++-savvy audiences
- Metaprograms that **scale** better
  → Better performing tools and more regular project management
- Makes metaprogramming **mainstream**
  → Brings more people to the table for contributing to amazing projects

However...

ctbench
compile time benchmarking for Clang

<< With great constexpr power comes great **compile time**. >>

Or not..?

ctbench
compile time benchmarking for Clang

Metaprogramming lacks tooling

Metaprogramming is *almost* on par with regular programing...

- ...but regular programming has debuggers, profilers,
- we know how to benchmark it to get **meaningful**, quantitative results,
- no such process for metaprograms,
- little to no **science** behind compile time rule of thumbs.

→ We need a sane process for understanding compile times.

ctbench
compile time benchmarking for Clang

How to measure compile times?

- Templight, *Zoltán Borók-Nagy, Zoltán Porkoláb, and József Mihalicza* (2009)
- Metabench, *Louis Dionne and Bruno Dutra* (2016)
- Build-Bench, *Fred Tingaud* (2017)
- Clang time-trace & Clang Build Analyzer, *Aras Pranckevičius* (2019)

ctbench
compile time benchmarking for Clang

Templight, *Zoltán Borók-Nagy, Zoltán Porkoláb, and József Mihalicza* (2009)

https://github.com/mikael-s-persson/templight

- Clang-based profiling tool, used as a drop-in replacement,
- profiles time and memory usage of **template instantiations**,
- interactive debugging,
- mostly a one-shot profiler,
- templight-tools provide several conversion options (Graphviz, XML, etc.)

→ Strong effort but no backend data

ctbench
compile time benchmarking for Clang

Metabench, *Louis Dionne and Bruno Dutra* (2016)

https://github.com/ldionne/metabench/

- Framework + compile time benchmarks for **libraries**,
- generates benchmarks of different sizes using Ruby templates (ERB),
- **black box** approach *(only measures compiler execution time)*,
- supports **all** compilers,
- web export

→ Valuable data, but harder to reuse

Build-Bench, *Fred Tingaud* (2017)

https://build-bench.com/

- Simple web interface,
- **black box** approach,
- supports **all** compilers,
- makes it easy to compare one-shot benchmarks,
- makes **per-compiler** comparisons possible

→ Nice to play with, but not extensive enough

ctbench
compile time benchmarking for Clang

Clang time-trace, *Aras Pranckevičius* (2019)

https://github.com/aras-p/llvm-project-20170507/pull/2

- **Code-aware profiler** provided by Clang,
- outputs flame graph **JSON** files for chrome://tracing,
- breaks down compile passes for **every symbol**,
- works for the frontend **and** the backend

→ Very strong reuse potential, available everywhere

ctbench
compile time benchmarking for Clang

Clang Build Analyzer, *Aras Panckevičius* (2019)

https://github.com/aras-p/ClangBuildAnalyzer

- Built on-top of **time-trace**
- Provides a user-friendly output

→ Extracts essential data from a time-trace file

ctbench
compile time benchmarking for Clang

What is ctbench?

https://github.com/jpenuchot/ctbench

- Compile time benchmarking & **data analysis** tool for Clang,
- built on-top of **time-trace**,
- **repeatability** & accuracy in mind,
- **variable size** benchmarks,
- C++ developer friendly:
  - C++ only benchmark files,
  - CMake API,
  - JSON config files (with a few ones already provided)

But how does it work?

ctbench
compile time benchmarking for Clang

Benchmarking methodology

- Benchmark set:
  - collection of benchmark cases to compare
- Benchmark case:
  - compilable C++ file,
  - compiled several times for a given **range of sizes**,
  - benchmark **iteration size** passed as a preprocessor define
- Benchmark iteration:
  - terminology for a benchmark case compiled with a **given size**,
  - **several samples** for each iteration size for improved accuracy
- Sample:
  - **one** time-trace file

→ Benchmark set → Benchmark cases → Benchmark iterations → Samples

ctbench

compile time benchmarking for Clang

Benchmark case - Recursive sum

```cpp
// Metaprogram to benchmark:
template <unsigned N> struct ct_uint_t { static constexpr unsigned value = N; };

template <typename T> auto sum(T const &) { return T::value; }
template <typename T, typename... Ts> auto sum(T const &, Ts const &...tl) {
  return T::value + sum(tl...);
}

// Benchmark driver:
#include <boost/preprocessor/repetition/enum.hpp>
#define GEN_MACRO(Z, N, TEXT) TEXT<N> {}
unsigned foo() {
  // return sum(ct_uint_t<1>{}, ..., ct_uint_t<BENCHMARK_SIZE>{});
  return sum(BOOST_PP_ENUM(BENCHMARK_SIZE, GEN_MACRO, ct_uint_t));
}
```

ctbench
compile time benchmarking for Clang

Benchmark case - Expansion sum

```
// Metaprogram to benchmark:
template <unsigned N> struct ct_uint_t { static constexpr unsigned value = N; };

template <typename... Ts> auto sum(Ts const &...) { return (Ts::value + ...); }

// Benchmark driver:
#include <boost/preprocessor/repetition/enum.hpp>
#define GEN_MACRO(Z, N, TEXT) TEXT<N> {}
unsigned foo() {
  // return sum(ct_uint_t<1>{}, ..., ct_uint_t<BENCHMARK_SIZE>{});
  return sum(BOOST_PP_ENUM(BENCHMARK_SIZE, GEN_MACRO, ct_uint_t));
}
```

→ *ctbench defines BENCHMARK_SIZE for each iteration size*

ctbench
compile time benchmarking for Clang

CMake API

- Benchmark declaration

```
ctbench_add_benchmark(variadic_sum.recursive  # Target name
  variadic_sum/recursive.cpp                  # Benchmark file
  1 32 1                                      # Start, stop, and step
  10)                                         # Number of repetitions
```

- Graph declaration

```
ctbench_add_graph(variadic_sum-graph  # Target name
  configs/feature_comparison.json     # Config file
  variadic_sum.expansion              # Benchmark target
  variadic_sum.recursive)             # ...
```

- Optional: Bring your own flags with ctbench_add_custom_benchmark

ctbench
compile time benchmarking for Clang

JSON configs

- Graph options:
  - plotter (currently: debug, simple curves, and stacked curves),
  - output file format (Gnuplot based thanks to Sciplot),
  - graph dimensions,
  - other plotter-specific graph options.
- Data options:
  - define named data groups,
  - implement predicates to target specific data,
  - predicates: regex matching, JSON inclusion matching, logical operators

→ Pre-made configurations available

ctbench

compile time benchmarking for Clang

Demo time!

ctbench
compile time benchmarking for Clang

Rule of Chiel

Ordering of compile time impact for several meta-programming techniques:

Expensive	SFINAE
	Instantiating a function template
	Instantiating a type
	Calling an alias
	Adding a parameter to a type
	Adding a parameter to an alias call
Cheap	Looking up a memoized type

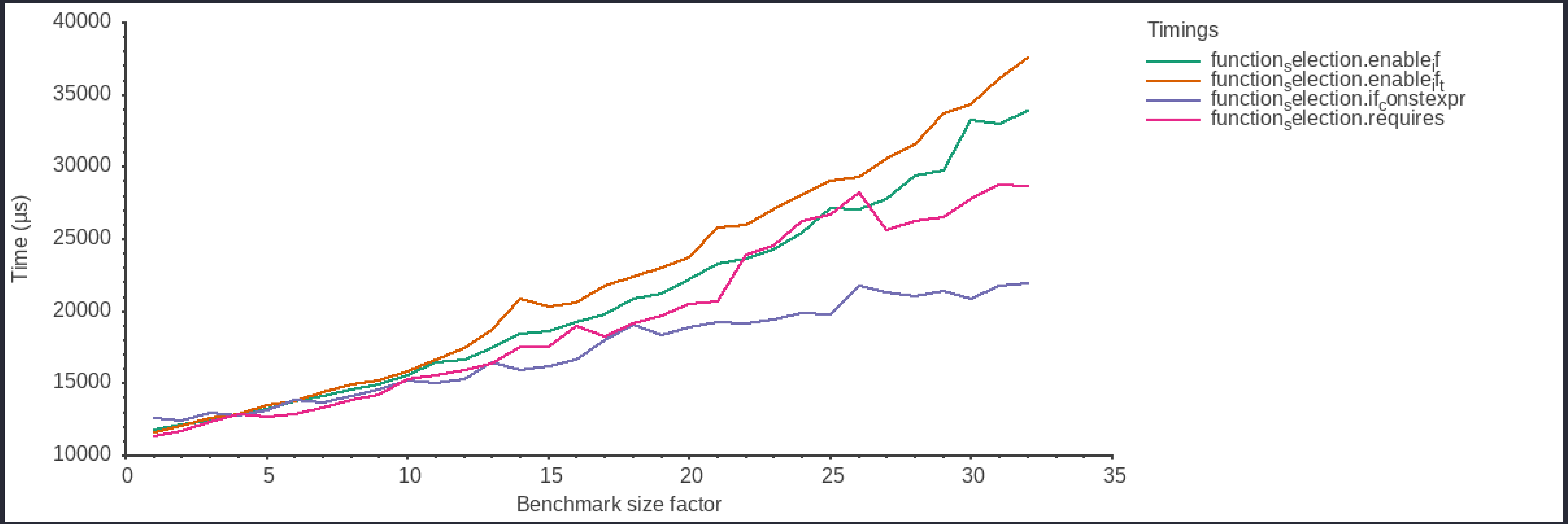
Source: "Type Based Template Metaprogramming is Not Dead", *Odin Holmes* (C++Now 2017)

→ Based on library implementation comparisons in **Metabench**

ctbench
compile time benchmarking for Clang

C++ contenders - Function selection

enable_if_t vs **enable_if** vs **if constexpr** vs **requires**

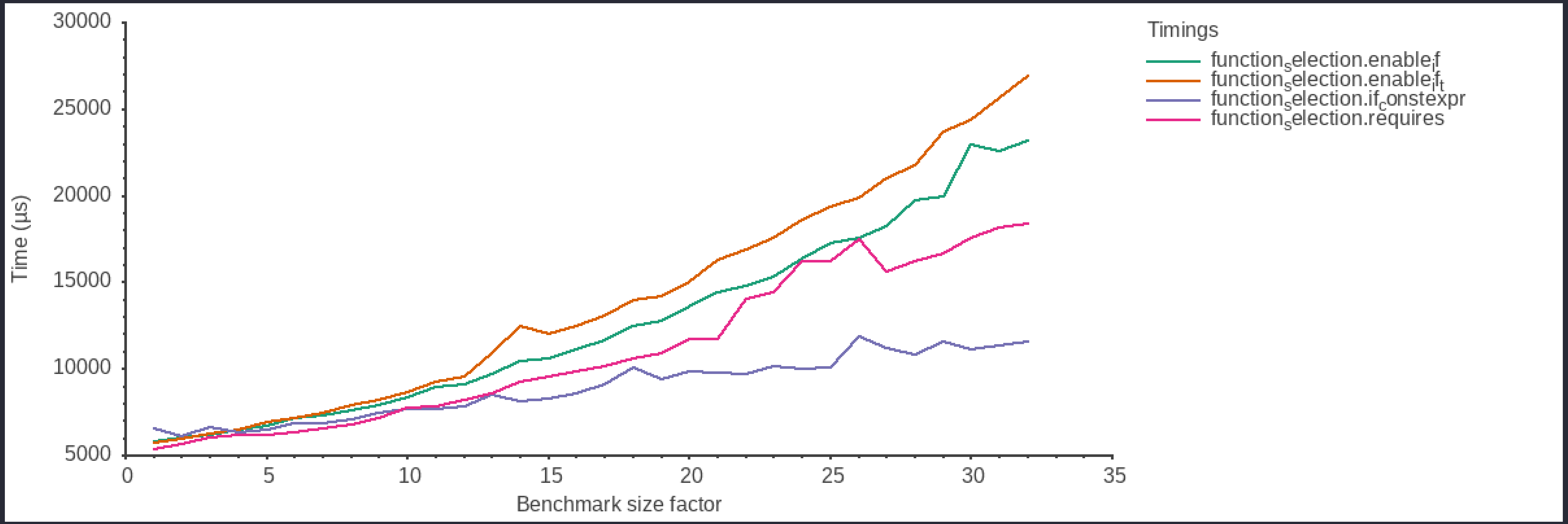


Targeted data: ExecuteCompiler

ctbench
compile time benchmarking for Clang

C++ contenders - Function selection

enable_if_t vs **enable_if** vs **if constexpr** vs **requires**

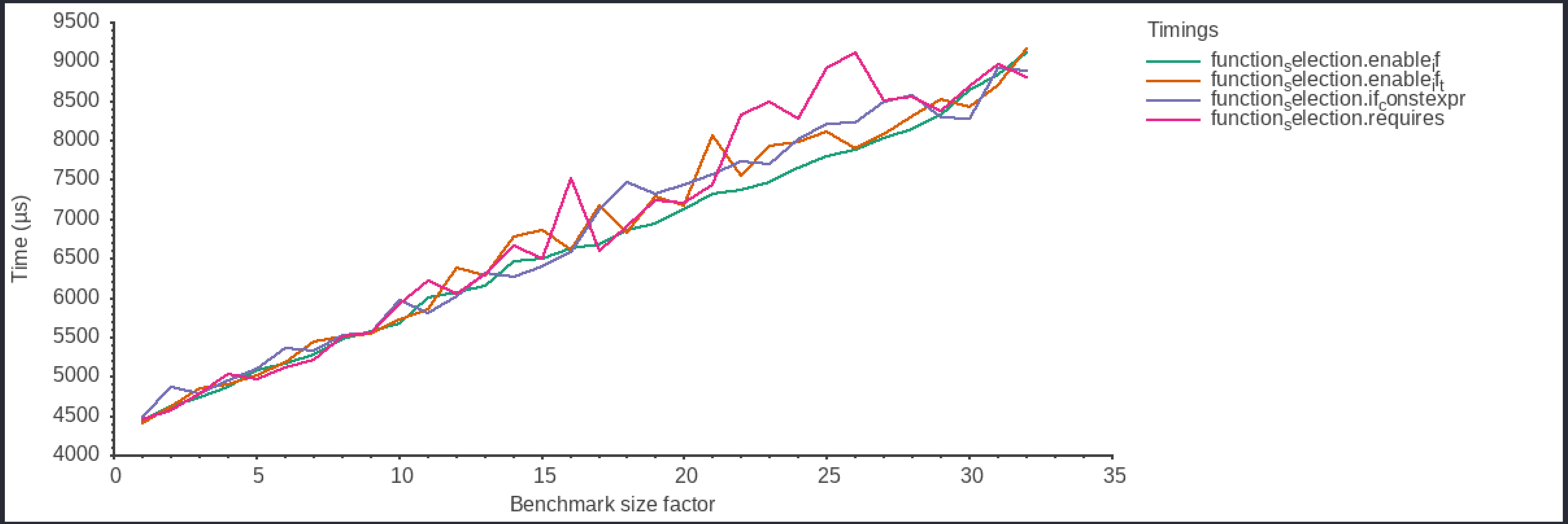


Targeted data: Frontend

ctbench
compile time benchmarking for Clang

C++ contenders - Function selection

enable_if_t vs **enable_if** vs **if constexpr** vs **requires**

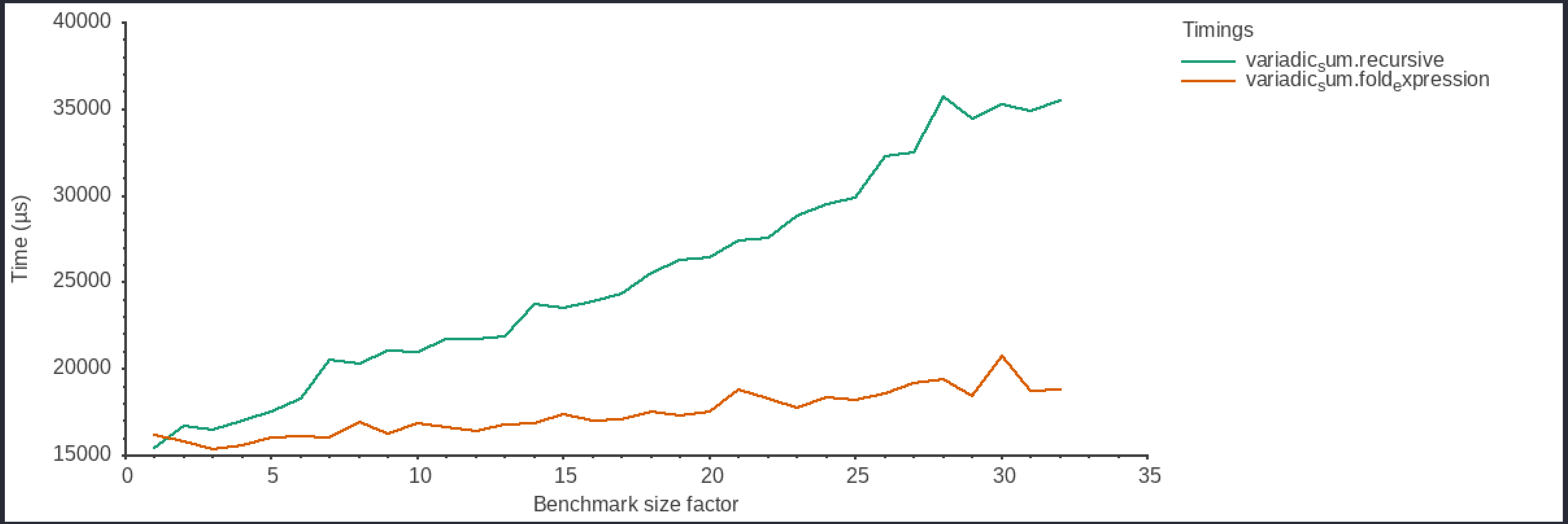


Targeted data: Backend

ctbench
compile time benchmarking for Clang

C++ contenders - Variadic sum

recursive vs **fold_expression**

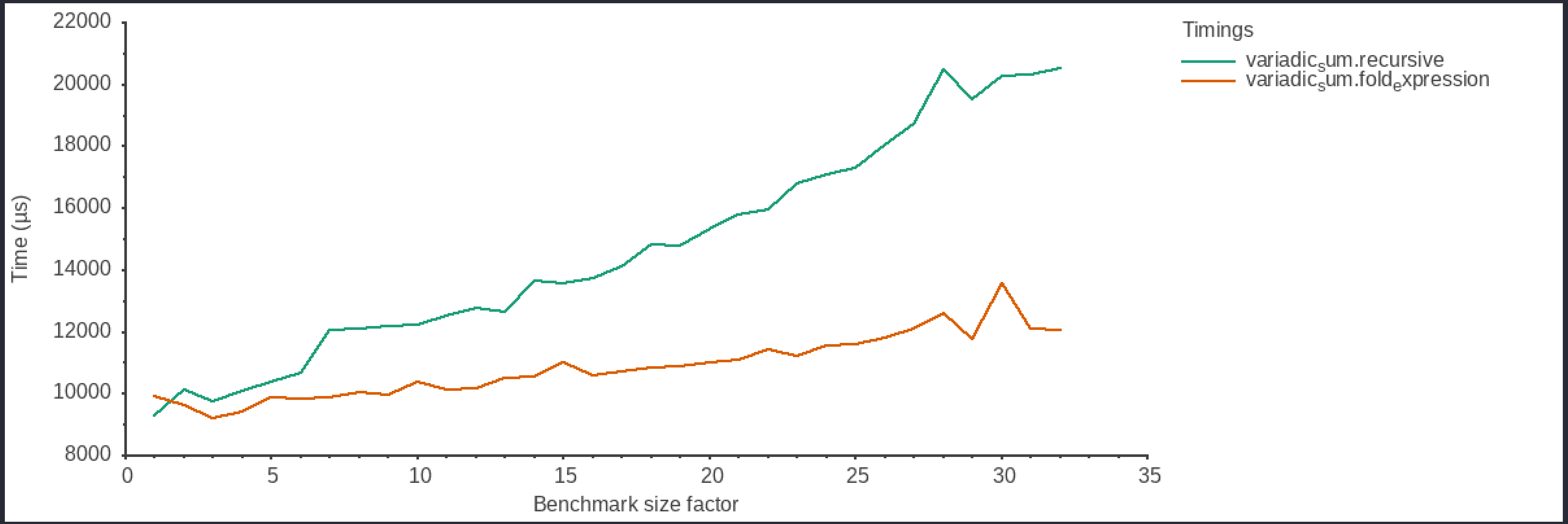


Targeted data: ExecuteCompiler

ctbench
compile time benchmarking for Clang

C++ contenders - Variadic sum

recursive vs **fold_expression**

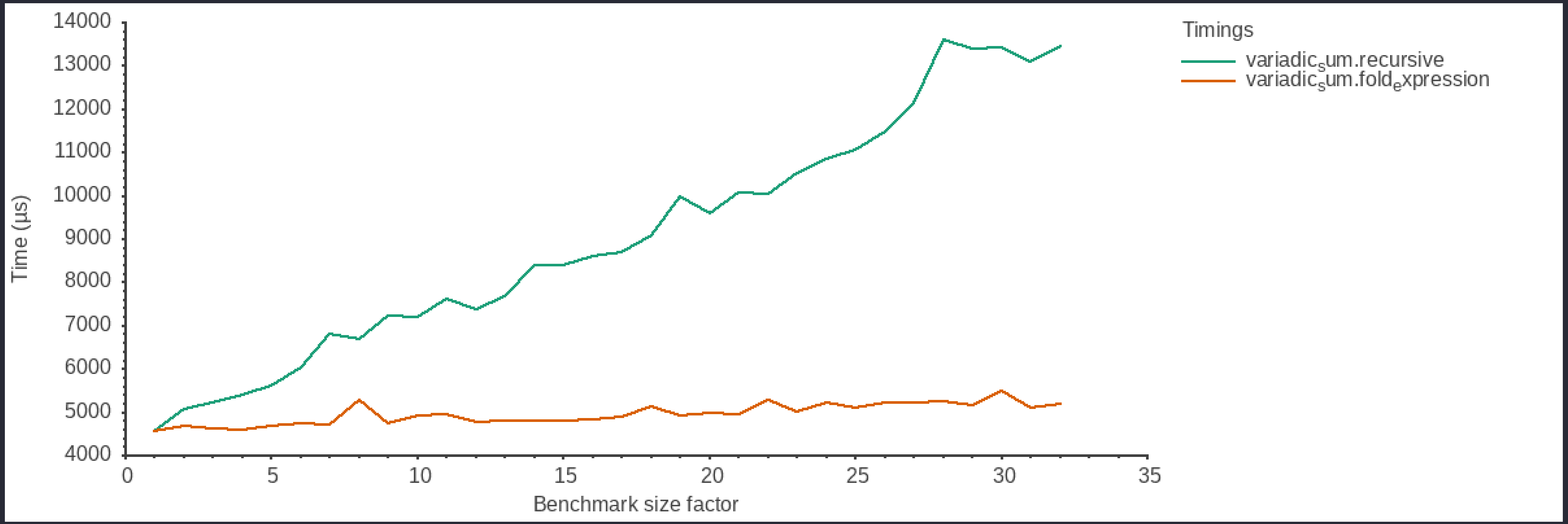


Targeted data: Frontend

ctbench
compile time benchmarking for Clang

C++ contenders - Variadic sum

recursive vs **fold_expression**



Targeted data: Backend

ctbench
compile time benchmarking for Clang

Project overview

- CMake API
  - **benchmarking.cmake** declares the end-user API,
  - documentation is provided inside (easily extracted into a MD file)
- **grapher** subproject (meatiest part)
  - CLI, time-trace file reading, predicate engine, and plotting,
  - designed as a **library** + CLI drivers,
  - relies heavily on **Sciplot** (https://github.com/Sciplot/Sciplot),
  - new plotters can be written easily
- Tooling:
  - **time-trace-wrapper**: clang exec wrapper to extract time-trace files
  - **cmake-doc-extractor**: extracts the API doc into a MD file

ctbench
compile time benchmarking for Clang

- What is the purpose of **ctbench**?

→ Better measurement and analysis tools where it's lacking, for a better **understanding** of compile times

- What do we have?

→ A flexible toolset to **run and analyze** Clang/LLVM compile-time benchmarks

- Is it perfect?

→ time-trace has **blindspots**, and we can't ignore **GCC**

ctbench
compile time benchmarking for Clang

Thank you!

https://github.com/jpenuchot/ctbench