CTBENCH - COMPILE-TIME BENCHMARKING

Jules Pénuchot

January 15, 2024

PhD Student Parallel Systems, LISN, Paris-Saclay University

Metaprogramming is evolving

Support libraries

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

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`

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 meta-programs,
- Little to no **science** behind compile time rule of thumbs.
- **We need a sane process for understanding compile times**.

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)

Introducing ctbench

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?

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

Benchmark case

Recursive sum

- *ctbench* defines `BENCHMARK_SIZE` for each iteration size.

```
1  // Metaprogram to benchmark:
2  template <unsigned N> struct ct_uint_t { static constexpr unsigned value = N; };
3
4  template <typename T> auto sum(T const &) { return T::value; }
5  template <typename T, typename... Ts> auto sum(T const &, Ts const &...tl) {
6    return T::value + sum(tl...);
7  }
8
9  // Benchmark driver:
10 #include <boost/preprocessor/repetition/enum.hpp>
11 #define GEN_MACRO(Z, N, TEXT) TEXT<N> {}
12 unsigned foo() {
13   // return sum(ct_uint_t<1>{}, ..., ct_uint_t<BENCHMARK_SIZE>{});
14   return sum(BOOST_PP_ENUM(BENCHMARK_SIZE, GEN_MACRO, ct_uint_t));
15 }
```

Benchmark case

Expansion sum

- *ctbench* defines `BENCHMARK_SIZE` for each iteration size.

```
// 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));
}
```

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)             # ...
```

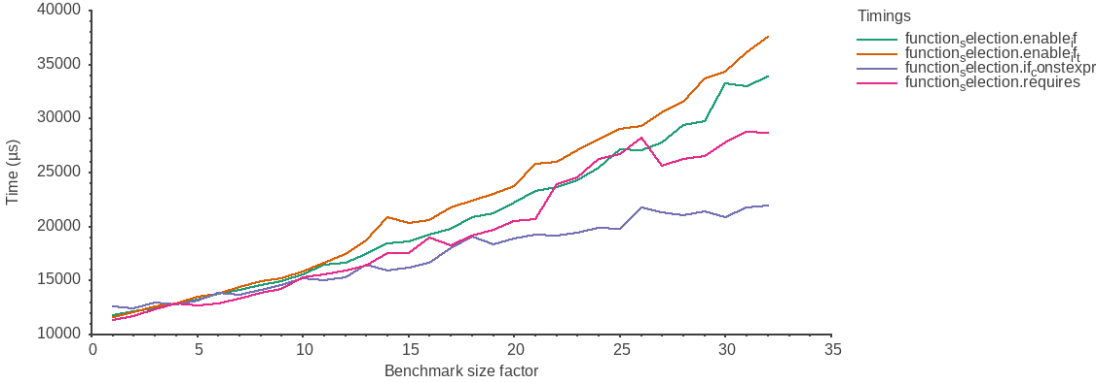
- Optional: Bring your own flags with `ctbench_add_custom_benchmark`

Sample benchmarks

C++ contenders

Function selection

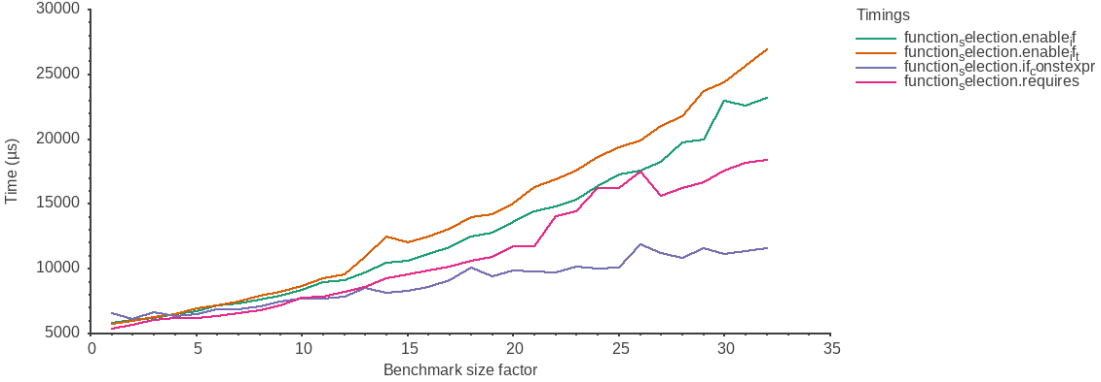
- enable_if_t, enable_if
- if constexpr
- requires



C++ contenders

Function selection

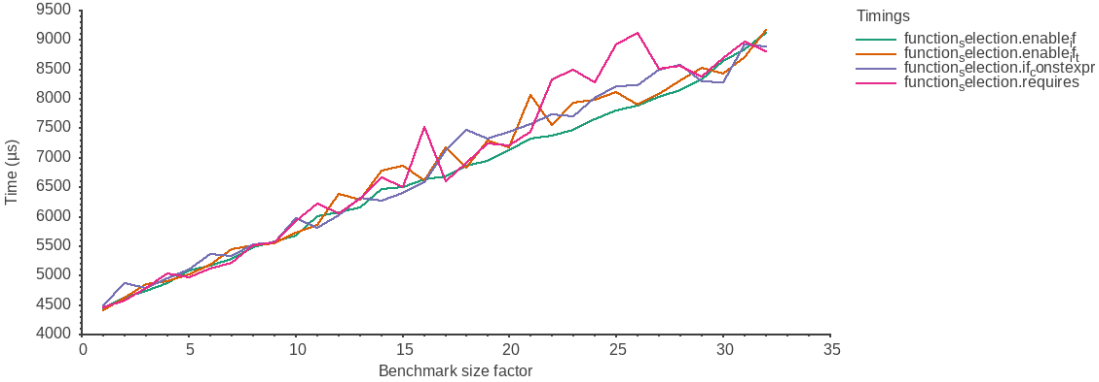
- **enable_if_t**, **enable_if**
- **if constexpr**
- **requires**



C++ contenders

Function selection

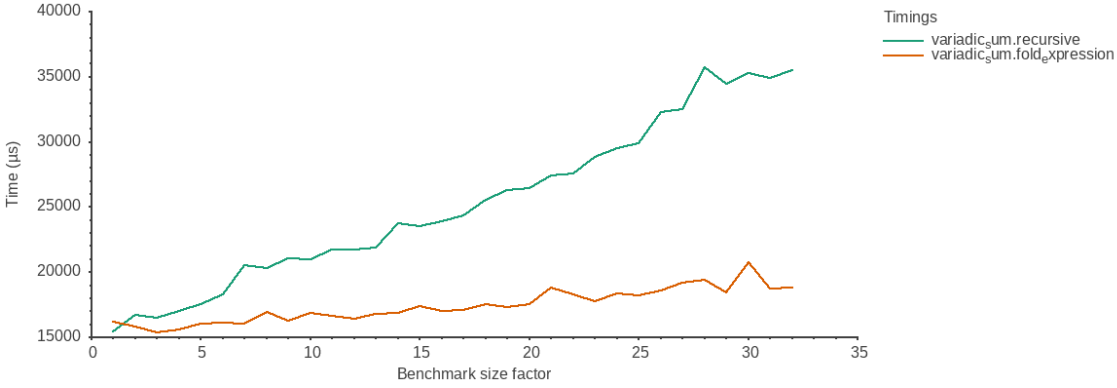
- enable_if_t, enable_if
- if constexpr
- requires



C++ contenders

Variadic sum

- recursive
- fold_expression

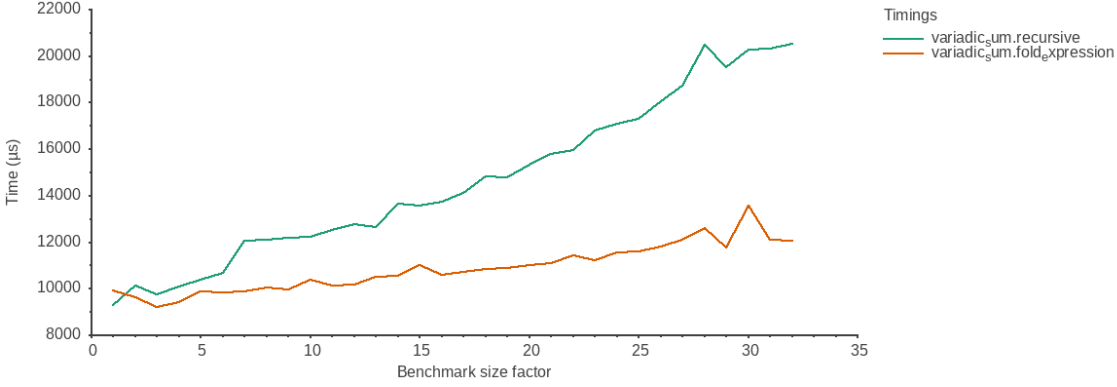


Targeted data: ExecuteCompiler

C++ contenders

Variadic sum

- recursive
- fold_expression

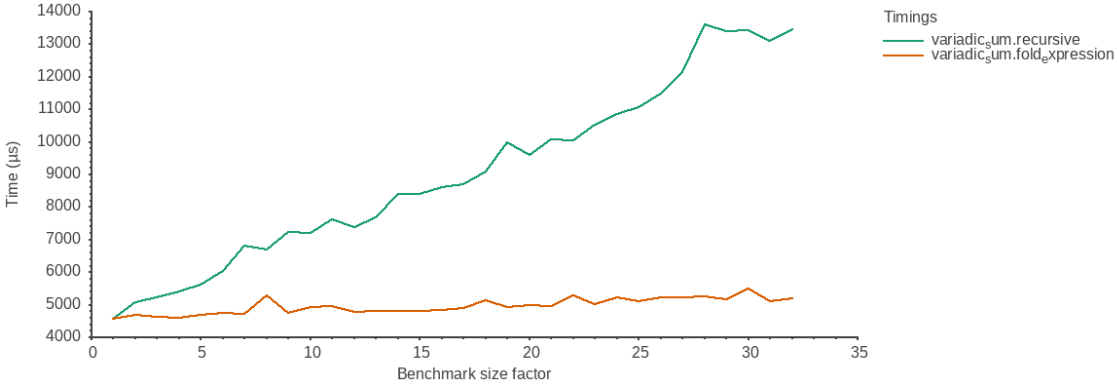


Targeted data: Frontend

C++ contenders

Variadic sum

- recursive
- fold_expression



Targeted data: Backend

Conclusion

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

Thanks for your attention !