Szybka prezentacja o przyspieszaniu budowania

Krzysztof Mazur, Gdańsk Embedded Meetup #23, 2024-10-08

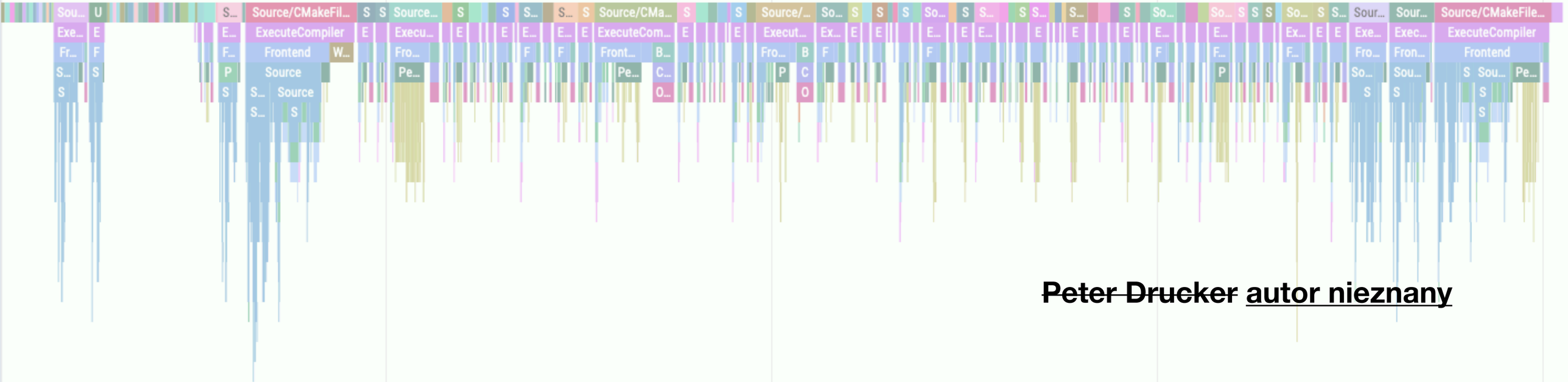
Budowanie

Czyli co?

1. Przygotowanie zależności
2. Konfiguracja procesu budowania
3. Kompilacja do plików obiektowych
4. Linkowanie
5. Testy?

https://xkcd.com/303/

“What gets measured, gets managed.”



~~Peter Drucker~~ autor nieznany

Zastrzeżenie

- Skupimy się na “czystym” budowaniu na lokalnej maszynie, choć część poruszonych tematów wpłynie także na inne scenariusze
- Pomijamy:
  - ccache - pamięć podręczna kompilacji, może być współdzielona
  - distcc - narzędzie do dystrybucji zadań kompilacji pomiędzy wiele maszyn

Ninjatracing

Od ogółu do szczegółu

- https://github.com/nico/ninjatracing
- Wymaga używania Ninja (🤯)
- Pozwala prześledzić kolejność i długość zadań kompilacji i linkowania
- Prosty w użyciu: `ninjatracing path/to/.ninja_log > trace.json`
- (demo) -> https://www.ui.perfetto.dev/

-ftime-trace

Zaglądamy głębiej

- Na razie tylko w clang
  - Nieoficjalny plugin dla GCC: https://github.com/royjacobson/externis
- Emituje informacje z każdej jednostki kompilacji
- `ninjatracing -e path/to/.ninja_log > trace_detailed.json`
- (demo)

Dobra, wiemy jak źle jest.
Co teraz?

Czego szukać

- Niepotrzebne zależności czasowe
  - Uruchomienie testów nie powinno czekać aż **wszystko** będzie zbudowane
  - https://cmake.org/cmake/help/latest/module/CMakeGraphVizOptions.html
- Niepotrzebna praca
  - Duplikacja kodu
  - https://github.com/include-what-you-use/include-what-you-use/blob/master/docs/WhyIWYU.md
- https://github.com/aras-p/ClangBuildAnalyzer

Precompiled headers

DRY czasu budowania

- Pliki nagłówkowe bywają zagnieżdżone i skomplikowane
  - Po co przetwarzać je więcej niż raz
- W CMake prosta opcja target_precompile_headers
- Wybieramy pliki do kompilacji na podstawie **danych**
- (demo)
- Każda zmiana jednego z wybranych plików nagłówkowych będzie wymagała przekompilowania wszystkich razem

SQL użyty w Perfetto do wybrania nagłówków

```
INCLUDE PERFETTO MODULE slices.slices;
select
    args.display_value,
    count(_slice_with_thread_and_process_info.arg_set_id) as cnt, sum(_slice_with_thread_and_process_info.dur)/1000000.0 as dur
from
    _slice_with_thread_and_process_info
    JOIN args ON args.arg_set_id = _slice_with_thread_and_process_info.arg_set_id
where
    _slice_with_thread_and_process_info.name = 'Source'
group by
    _slice_with_thread_and_process_info.arg_set_id
order by
    dur desc
limit
    20;
```

Przykładowe wyniki

Na przykładzie https://github.com/google/bloaty

- **30% przyspieszenia**
  - Przed: średnio 37,286s, po: średnio 25,879s
- 10 prekompilowanych plików nagłówkowych…
  - Skompilowanych do pliku o rozmiarze 39MB

Alternatywy

- Make: 😔
- GCC -> https://github.com/royjacobson/externis (?)
- bazel: https://bazel.build/advanced/performance/json-trace-profile

Rust 🦀

- `cargo build --timings`
- https://doc.rust-lang.org/cargo/reference/timings.html

Źródła

- https://aras-p.info/blog/2019/01/16/time-trace-timeline-flame-chart-profiler-for-Clang/
- https://www.snsystems.com/technology/tech-blog/clang-time-trace-feature

Dziękuję za uwagę