

Profile-guided optimization

Speeding up LHCb software through compilation optimization

Oscar Buon

June 20, 2023

Supervisor: Sebastien Ponce

CERN

Intergovernmental particle physics on France-Switzerland border.
About 15,000 people working at CERN.
Made discoveries that led to Nobel Prizes.
World Wide Web was invented at CERN.



Large Hadron Collider (LHC)

- Large Hadron Collider: particle collider
- 27km (biggest in the world)
- $\sim 100m$ underground



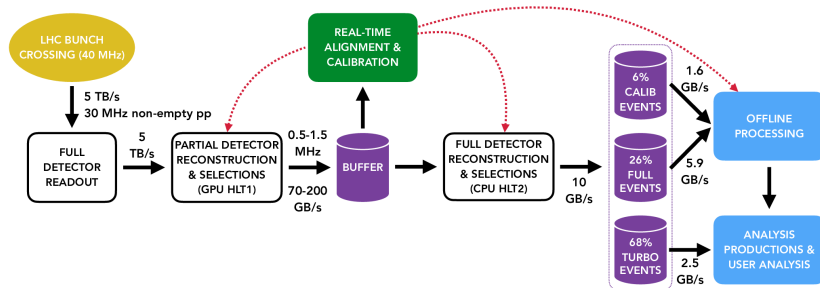
LHCb

- One of the 4 main experiments installed on the LHC.
- More than 1,200 people working for the collaboration.
- Studies asymmetry between matter and antimatter via b-physics. Collision of hadrons (heavy particles).

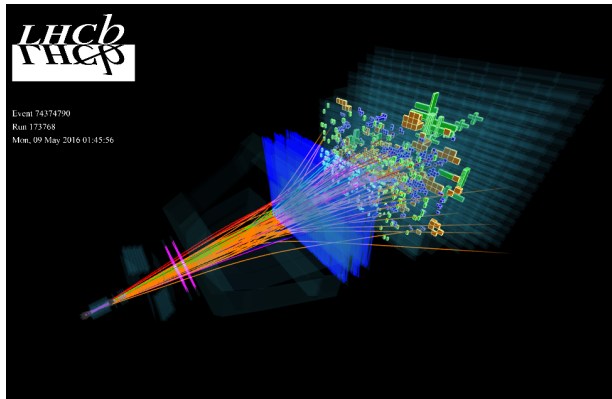


Physics Computing

Important computing infrastructure.



Computing group



The computing group has to maintain this infrastructure for LHCb.

Software

The software is a stack of programmes:

- Common base shared with other experiments.
- LHCb specific programmes.
- Reconstruction, simulation...

Millions lines of code which some are 30 years old. Code mainly written by non-software engineers.

All of this is running on thousands of multithreaded Linux servers.

Table of contents

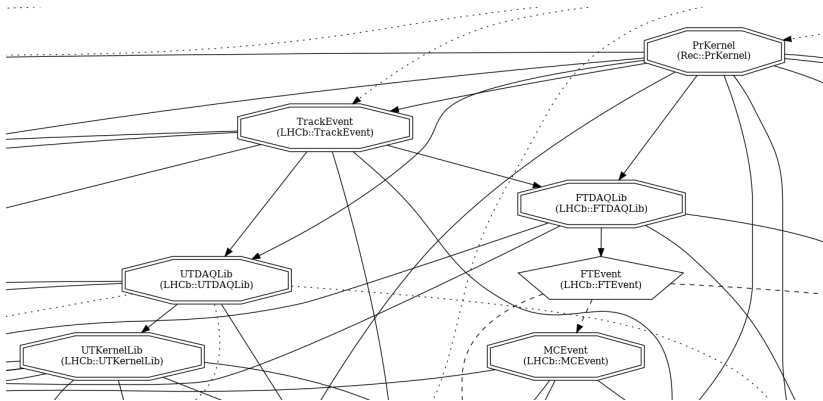
- 1 Graphviz for printing graphs
- 2 Static libraries and modules
 - Library types
 - Static libraries
 - Static modules
- 3 Profile-guided optimization
 - Profile-guided optimization
 - Link-time optimization
 - Final building pipeline

- 1 Graphviz for printing graphs
- 2 Static libraries and modules
 - Library types
 - Static libraries
 - Static modules
- 3 Profile-guided optimization
 - Profile-guided optimization
 - Link-time optimization
 - Final building pipeline

Graphviz for printing graphs

Printing graphs was usefull to have an idea of the dependencies.
Graphviz in CMake to create dependency graphs:

- CMake arg: `--graphviz=path/to/files.dot`
- Convert to SVG:
`dot -Tsvg -o path/to/file.svg path/to/file.dot`
- Make arg: `CMAKEFLAGS="--graphviz=path/to/files.dot"`
- Setting `GRAPHVIZ_EXTERNAL_LIBS` to `FALSE` is usefull.



- 1 Graphviz for printing graphs
- 2 Static libraries and modules
 - Library types
 - Static libraries
 - Static modules
- 3 Profile-guided optimization
 - Profile-guided optimization
 - Link-time optimization
 - Final building pipeline

Library types

- **STATIC**: archive of object files that are merged to the executable at link time.
- **SHARED**: `.so` (Linux) file that is automatically linked to the executable at runtime (when starting it). Allows several executables to link with a library installed on the system.
- **MODULE**: same as shared but the module is loaded on demand via the `dlopen` function, so not automatically.

Gaudi Static libraries

- Gaudi based on CMake.
- Created new Gaudi function for static libraries:
 - Adapting the shared one.
 - `gaudi_add_library` \Rightarrow `gaudi_add_static_library`
 - Replacing SHARED by STATIC in `add_library` function call.

Static modules

Linking modules as static libraries requires:

- `-Wl,--whole-archive`
 - without the linker removes what seems to be unused code.
- `-Wl,--allow-multiple-definition`
 - `-Wl,--whole-archive` causes symbols to be included several times.
- `-Wl,--export-dynamic`
 - allow functors to access to symbols.

These flags are not recommended. There may still be bugs, especially with functors.

Other Limitations of the Prototype

- The final executable goes from $\sim 20kB$ to $2.5GB$ (opt+g).
- Standard method to run test with python not working.
 - Need to run directly the executable with a json options file:

```
build/run ./build/Gaudi/Gaudi/Gaudi_static options.json
```

- 1 Graphviz for printing graphs
- 2 Static libraries and modules
 - Library types
 - Static libraries
 - Static modules
- 3 Profile-guided optimization
 - Profile-guided optimization
 - Link-time optimization
 - Final building pipeline

Profile-guided optimization

The compiler uses some heuristics to optimize some elements:

- Inlining
- Block ordering
- Register allocation
- Virtual call speculation
- Dead code separation

Makes programmes running several times faster. But a few times these heuristics are wrong.

A better way for the compiler should be knowing running behavior.
The the principle of PGO is:

- Compiling the programm with instrumentation.
- Running it to create profiles (counters).
- Recompiling the programm with the profiles.

Link-time optimization

- C/C++ program composed of translation units
 - 1 translation unit \approx 1 .c/.cpp file.
- Compiler cannot optimize accross them by default.
- LTO allows the linker to perform optimizations that take account of all translation units.

Final building pipeline

- Compiling the program with `-fprofile-generate`
- Running it to create profiles
- Recompiling the program with
`-flto -fprofile-use -fprofile-correction`

Conclusion

Test: hlt2_pp_thor (6×1000 events)

Optimization	Improvement	Confidence interval (2σ)
LTO	0.17%	$\pm 1.12\%$
LTO & PGO	6.74%	$\pm 1.44\%$
Static LTO	0.87%	$\pm 0.60\%$
Static LTO & PGO	6.88%	$\pm 0.83\%$

- Using LTO & PGO makes the programm running faster.
- Using static libraries and modules doesn't seem to be usefull and leads to some bugs.

Appendix

- Stack:

`https://gitlab.cern.ch/clemenci/lhcb-super-project-template/`

- PGO: `https://gitlab.cern.ch/obuon/lhcb-pgo`

- Target: `x86_64_v3-centos7-gcc11+detdesc-opt+g`