Testing AutoFDO for Geant4

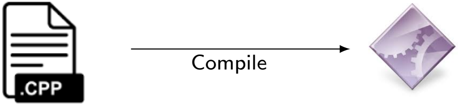
Nathalie Rauschmayr

IT-CF-FPP

With help from Benedikt Hegner and Shahzad Malik Muzaffar

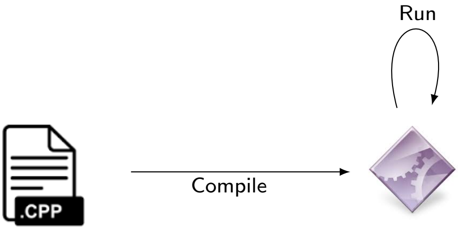
Introduction

Idea: Autotuning



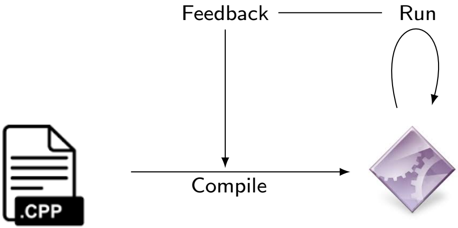
Introduction

Idea: Autotuning



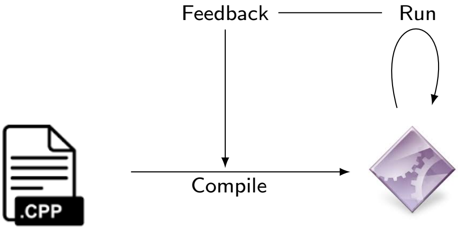
Introduction

Idea: Autotuning



Introduction

Idea: Autotuning



Concept exists already for some time: **Profile Guided Optimization**

Introduction

Why it helps to improve performance:

Introduction

Why it helps to improve performance: LHC code consists of a lot of branches/dependencies

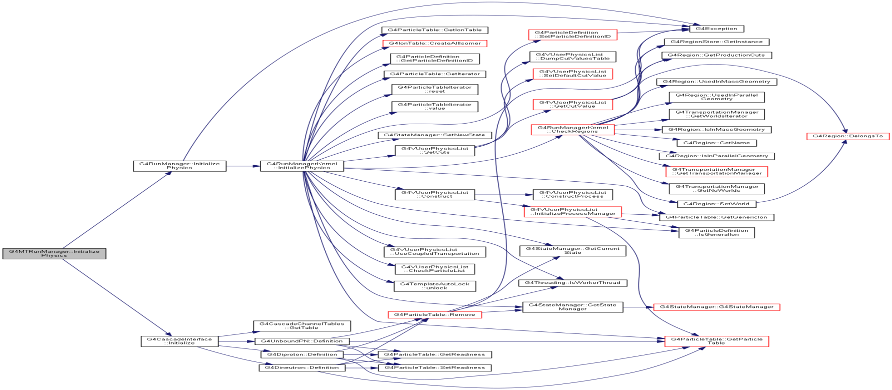


Figure: Example from Geant4: G4MTRunManager::InitializePhysics()

Introduction

Profile Guided Optimization is useful for:

- Code that contains a lot of branches that are difficult to predict at compile time
- Performance sensitive code
- When running the same code over and over again

Introduction

Profile Guided Optimization:

- Uses profiling to improve runtime performance
- Analyses code sections that are frequently executed
- Based on profiles the compiler might change:
    - Inlining
    - Virtual Call Speculation
    - Register allocation
    - Basic Block Optimization
    - Function Layout
    - Conditional Branch Optimization
    - Dead Code Separation

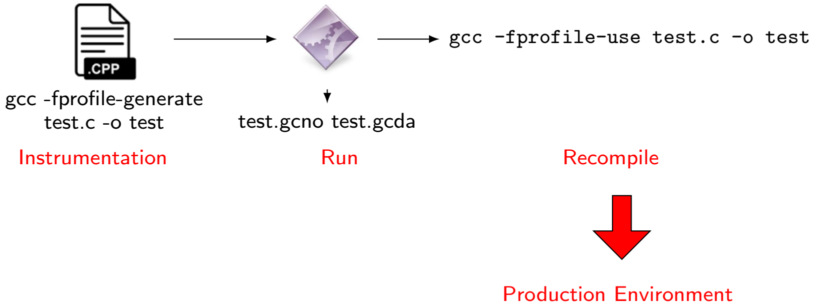
Introduction

Two approaches for Profile Guided Optimization (PGO):

- Modify binary (instrumentation)
- Monitor unaltered binary (sampling with perf)
  - AutoFDO transforms perf-profiles into the format that can be used by gcc/clang for Feedback Directed Optimization (FDO)
  - Developed by Google `https://github.com/google/autofdo`

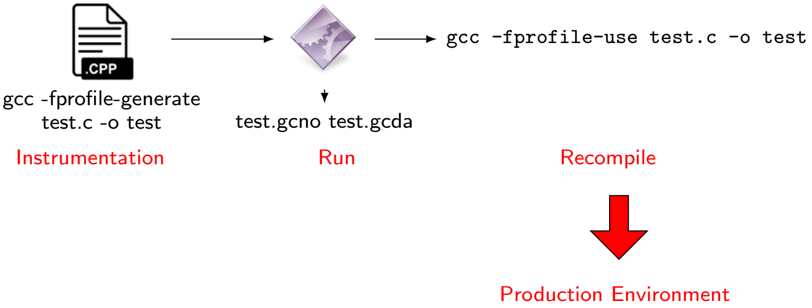
Difference between sampling and instrumentation

Instrumentation based PGO:



Difference between sampling and instrumentation

Instrumentation based PGO:

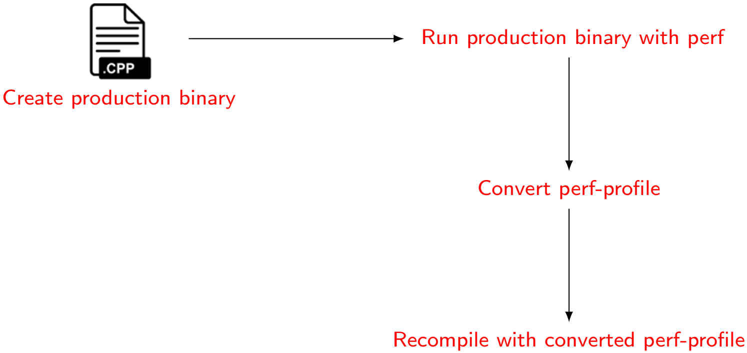


Disadvantages:

- Tedious dual-compilation
- Produces a lot of small output files (in case of Geant4: 1698 files, each smaller than 100KB)
- Cannot run easily in production environment
- Instrumented binary might be significantly slower

Difference between sampling and instrumentation

Sampling Based FDO (AutoFDO):



Difference between sampling and instrumentation

Sampling Based FDO (AutoFDO):

gcc -O3 -ggdb
-frecord-compilation-info-in-elf
-D_DEBUG test.c -o test

Create production binary

perf record -b -e
cpu/event=0xc4,umask=0x20,
name=br_inst_retired_near_taken,
period=1000009/pp ./test

Run production binary with perf

create_gcov --binary=./test
--profile=perf.data --gcov=binary.gcov
-gcov_version=1

Convert perf-profile

gcc -O3 -fauto-profile=test.gcov
test.c -o test

Recompile with converted perf-profile

Difference between sampling and instrumentation

AutoFDO compared to instrumentation based PGO:

- Profile data can be obtained in production environment
- Works on optimized builds
- It provides a tool to merge profiles from multiple runs
- Only one output file per run

General Caveats

- The sample needs to be representative for the typical usage scenarios
- Otherwise: PGO could possible slow down the performance
- Need many profiles and runs
- Unbiased branches

Testcases

Applications:

- CMS Detector Simulation (FullCMS)
- Simulation step of CMSSW using static build of Geant4 (cmsRun)

Input data/workflow needs to be representative:

- How many events needed as training data?
- What if job configuration changes?
- What if job type changes?

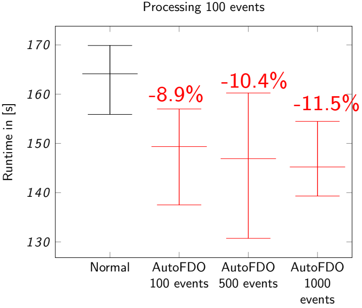
Testcases

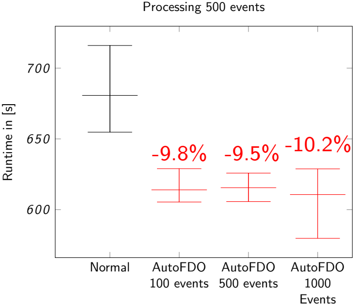
Training data	Run	Number of Events
FullCMS run	FullCMS run	100,500,1k
cmsRun config1	cmsRun config1	20, 50, 100
cmsRun config1	cmsRun config2	20, 50, 100
cmsRun config2	cmsRun config2	20, 50, 100
FullCMS run	cmsRun config2	1k

FullCMS: Geant4 example with particle gun
cmsRun config1: TTbar event generation and simulation (CMSSW_7_3_1)
cmsRun config2: Wjets event generation and simulation (CMSSW_7_3_1)

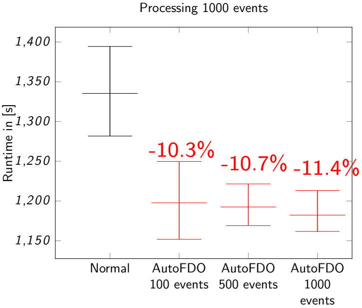
CMS Full Detector Simulation

Training data	Run	Number of Events
FullCMS run 100 events	FullCMS	100, 500, 1k
FullCMS run 500 events	FullCMS	100, 500, 1k
FullCMS run 1k events	FullCMS	100, 500, 1k





CMS Full Detector Simulation



Simulation step of CMSSW using BigProducts

Used CMSSW_7_3_1:

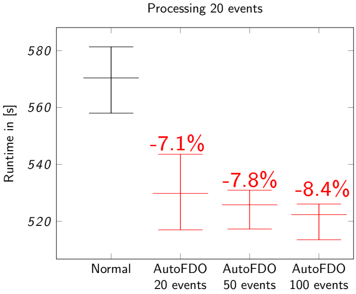
- SLC6, kernel 3.16
- gcc 4.8
- It uses BigProducts by default (developed by Shazhad)
  - pluginSimulation.so: linked against static Geant4 libraries
  - Obtain perf-profile for cmsRun, but then optimize only pluginSimulation.so

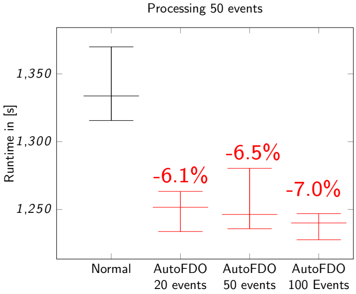
Testcase: TTbar

- Step 1: Event generation and simulation
- 20, 50, 100 events

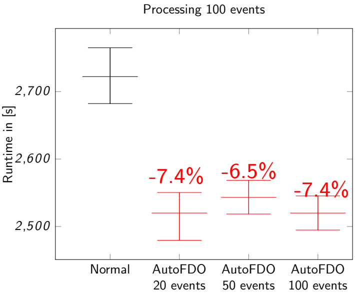
Simulation step of CMSSW using BigProducts

Training data	Run	Number of Events
cmsRun 20 events config1	cmsRun config1	20, 50, 100
cmsRun 50 events config1	cmsRun config1	20, 50, 100
cmsRun 100 events config1	cmsRun config1	20, 50, 100





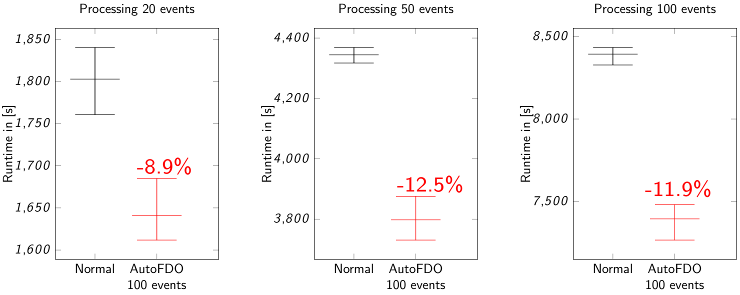
Simulation step of CMSSW using BigProducts



Simulation step of CMSSW using BigProducts

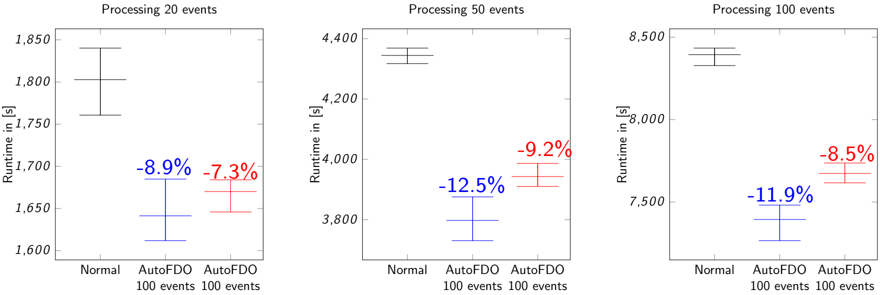
cmsRun config2: took Pythia configurations from Wjet_Pt_3000_3500_14TeV_cfi.py in CMSSW_8_1_X

Training data	Run	Number of Events
cmsRun 100 events config1	cmsRun config2	20, 50, 100



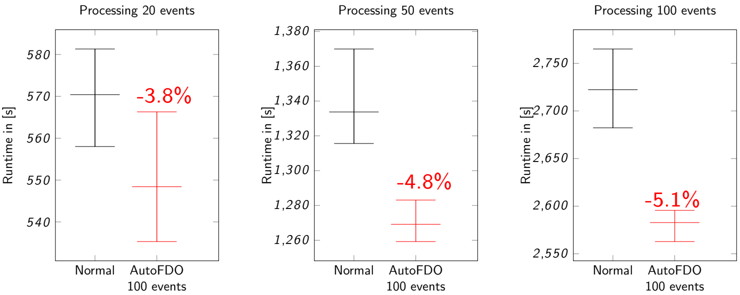
Simulation step of CMSSW using BigProducts

Training data	Run	Number of Events
cmsRun 100 events config1	cmsRun config2	20, 50, 100
cmsRun 100 events config2	cmsRun config2	20, 50, 100



Simulation step of CMSSW using BigProducts

Training data	Run	Number of Events
fullcms 100 events	cmsRun job config2	20, 50, 100



AutoFDO gives useful insight

Google-gcc provides the flag `-fcheck-branch-annotation`:

```
objcopy -O binary --set-section-flags .gnu.switches.text.branch.annotation=alloc
-j .gnu.switches.text.branch.annotation  libG4processes.so  libAnnotated
```

Example Output:

G4EnhancedVecAllocator.hh;122;146;0;10000;9550;d9a18bb69d5efaf3d9068625ec56d66a
G4EnhancedVecAllocator.hh;137;8389;0;225;450;6a740d527b3f213d4868919fc7d9710c
G4EnhancedVecAllocator.hh;135;8389;0;10000;9550;a17d8feb82daee40febb118864576dc9

AutoFDO gives useful insight

Google-gcc provides the flag `-fcheck-branch-annotation`:

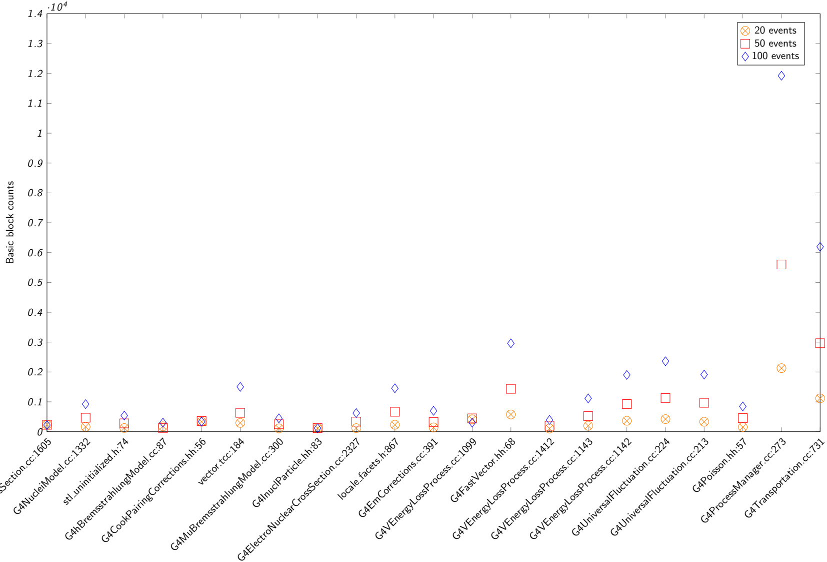
```
objcopy -O binary --set-section-flags .gnu.switches.text.branch.annotation=alloc
-j .gnu.switches.text.branch.annotation  libG4processes.so  libAnnotated
```

Example Output:

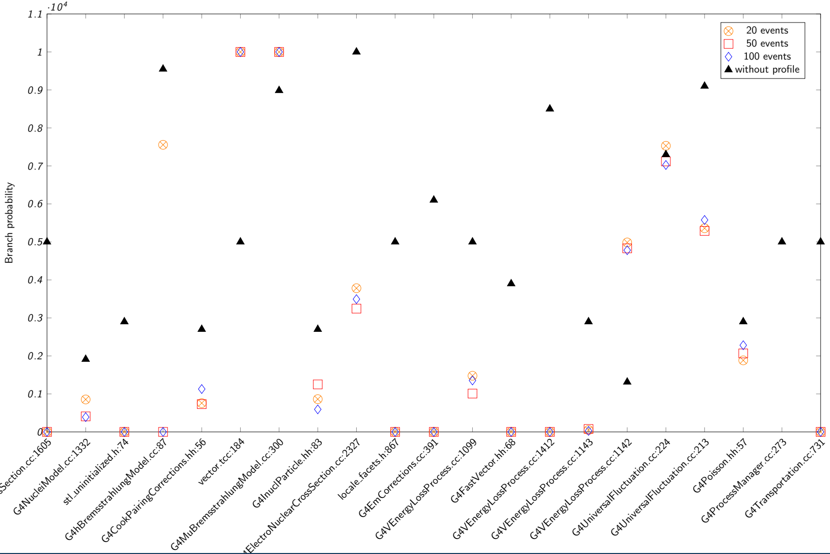
G4EnhancedVecAllocator.hh;122;146;0;10000;9550;d9a18bb69d5efaf3d9068625ec56d66a
G4EnhancedVecAllocator.hh;137;8389;0;225;450;6a740d527b3f213d4868919fc7d9710c
G4EnhancedVecAllocator.hh;135;8389;0;10000;9550;a17d8feb82daee40febb118864576dc9

- File
- Line
- Basic block count
- Annotated
- Measured branch probability
- Assumed branch probability
- Hash value

AutoFDO gives useful insight



Compiler heuristics are not always accurate for branch probabilities



Summary

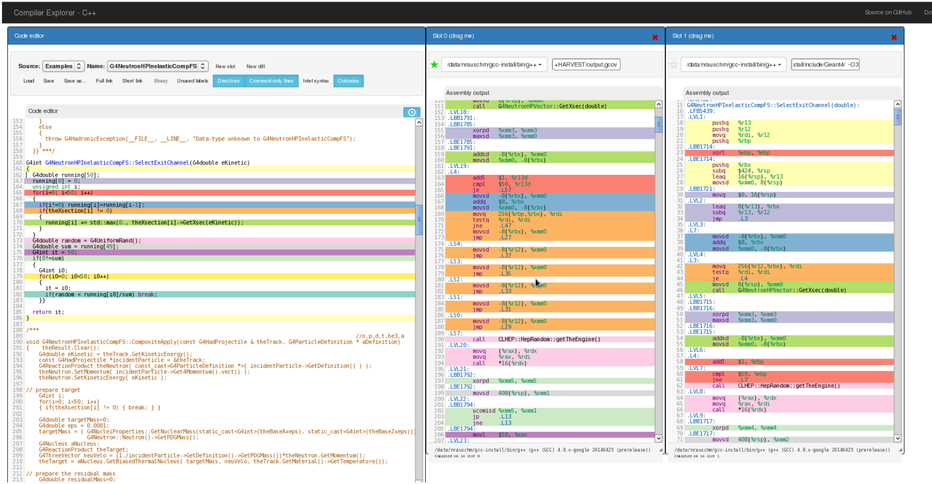
- Quite decent speedups: 5-13%
- Some fixes needed (shared libraries, gcc-flag)
- Stable against change of simulation scenario
- Easy deployment
    1. Start perf together with the job
    2. Gather profiles
    3. Convert and merge profiles
    4. Add compiler flag in CMake scripts

Backup Slides

Applied Optimizations

Perf profile delivered as number one hotspot:

G4NeutronHPInelasticCompFS::SelectExitChannel with 5.9 % of br_inst_retired_near_taken



Applied Optimizations

Information from gcc:

```
gcc  -fauto-profile=/data/nrauschm/CMSSW_7_3_1/output.gcov -fopt-info-optimized
```

```
G4NeutronHPInelasticCompFS.cc:182:5: note: Unroll loop 9 times
G4NeutronHPInelasticCompFS.cc:168:3: note: Unroll loop 6 times
```

Problems encountered

- Google gcc 4.8 has been merged with gcc main branch
- But: normal gcc is missing the flag -frecord-compilation-info-in-elf
- Flag creates a new section header and records compiler command line options

```
>>>readelf -S fullcms | less
There are 46 section headers, starting at offset 0x176209c0:
Section Headers:
[Nr] Name              Type             Address           Offset
     Size              EntSize          Flags  Link  Info  Align
[ 0]                   NULL             0000000000000000  00000000
     0000000000000000  0000000000000000           0     0     0
[ 1] .note.ABI-tag     NOTE             0000000000400190  00000190
     0000000000000020  0000000000000000   A       0     0     4
[...]
[29] .gnu.switches.tex PROGBITS         0000000000000000  01712830
     0000000000a00569  0000000000000000           0     0     1
```

Problems encountered

- The information in .gnu.switches.text is used to build the module map
- create_gcov dumps the module map to a file (ending with .imports)

```
>>>head  output.gcov.imports
/data/geant4.10.01.p03/source/event/src/G4EventManager.cc: /data/geant4.10.01.p03/
/data/geant4.10.01.p03/source/event/src/G4SmartTrackStack.cc: /data/geant4.10.01.p0
/data/geant4.10.01.p03/source/event/src/G4StackManager.cc: /data/geant4.10.01.p03/
/data/geant4.10.01.p03/source/externals/clhep/src/Evaluator.cc: /data/geant4.10.01.
/data/geant4.10.01.p03/source/externals/clhep/src/LorentzRotation.cc
/data/geant4.10.01.p03/source/externals/clhep/src/LorentzVector.cc
/data/geant4.10.01.p03/source/externals/clhep/src/LorentzVectorL.cc
```

Problems encountered

- Apart from the module map AutoFDO creates a symbol map
- Symbol map does not contain symbols coming from shared libraries
- It limits the usability:
  - Statically linked libraries required
  - Optimize only the library causing the largest hotspots
- icc-files are not recognized (fixed)
- recent versions of perf could be problematic because data format is different
- works best with sampling the Last Branch Records (LBR)
  - LBR: Collection of register pairs that store source and destination addresses of recently executed branches (currently only Intel CPUs)