GEOS / GCC-8 Ubuntu Linux Profiling & Instrumentation Notes

darkblue_b / March 2019 / maplabs -@- light42 . com

INTRODUCTION

With new code and construction ahead, **libGEOS 3.8.0dev** makes a complex testing problem. This document will detail a few approaches to measuring, monitoring and debugging ELF format binaries generated with **gcc** toolchains on linux.

TOOLCHAIN

gcc-8 ## installs 8.1x at the time of writing
gdb-8.2 ## from source **`gdb-8.2.tar.gz`**
valgrind-3.14.0 ## from source **`valgrind-3.14.0.tar.bz2`**

```
#!/bin/bash

##-- Add gcc8x to Ubuntu 1604 xenial (.. from a popular gist)
sudo apt-get update -y &&
sudo apt-get install build-essential software-properties-common -y &&
sudo add-apt-repository ppa:ubuntu-toolchain-r/test -y &&
sudo apt-get update -y &&
sudo apt-get install gcc-8 g++-8 -y &&
sudo update-alternatives \
  --install /usr/bin/gcc gcc /usr/bin/gcc-8 60 \
  --slave /usr/bin/g++ g++ /usr/bin/g++-8 &&
sudo update-alternatives --config gcc

select gcc-8
```

Build for Testing Workflows

A source code base in `git` defines the commit IDs, tags and branches of a changing project.
The following are parts of an application building pipeline, emphasizing testing.

Build Target Binary Executables : defined via Cmake and/or autotools; recent builds show
`astyle test_bug234 test_geos_unit test_simplewkttester test_sweep_line_speed test_xmltester`

Library targets include :
`libgeos_c.so.1.11.0` `libgeos.so.3.8.0dev` `libgeos.a`

Source Code built : git (tag,branch,remote) triple defines the source input

CXXFlags Settings : in order to record a set of build instructions, including `CXXFLAGS` and linking, a convention is established to keep a `do_build_uniqname_vers.sh` script in the same directory as a subject test app source file.
It is up to the testing on a case-by-case basis to build `libgeos` freshly, or not.

Libraries linked : libraries that are hard requirements for GEOS to run, are largely supplied at the OS system package level. Sources of OS installable packaging (`.deb`) on a testbed Ubuntu Linux include:
GNU/Linux Debian Canonical/Ubuntu pgdg Postgres UbuntuGIS selected third parties

CORE LIBRARY origin, packaged name and version installed, could be recorded in sets for testing recreation.
Fortunately it is relatively easy to get the fully qualified paths to linked libraries, either statically on disk or at runtime.

App Executables : profiling and debugging are straightforward on a finite application run, a single binary; the execution being measured has a start, a middle and a clean end. Shared libraries in practice are often used as long-lived services. So profiling and debugging have new challenges.
It is useful to create small test applications that can be run under very controlled conditions, for profiling and debugging.

In order to create comparisons of test conditions in an orderly way, it is desirable to record the **four attributes** shown above, for any test app. What executable ; What source code base ; Which linked libraries ; What compiler/link flags .

Note for more consistant testing results, test-app builds here
assume the x86_64 environment and explicitly enable Intel AVX support
see **AVX** https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

```
CXXFLAGS   -march=sandybridge  -mtune=sandybridge
```

TOOL DETAIL

callgrind + kcachegrind ----------

enable by compiling with any 'dash-g' option in `CXXFLAGS` .. for example `-Og -ggdb3`

callgrind is a **Valgrind** tool for "call graph" profiling programs.
Note that callgrind profiling is generally performant enough to run on production installed binaries.
Targets built with full optimizations work well with callgrind. `CXXFLAGS -O2`

Source code does not have to be modified in order to use callgrind+kcachegrind

see *valgrind-3.14.0/callgrind/docs*

callgrind is for *performance profiling*; see below for *code coverage*

Invocation:

```
valgrind tool=callgrind  app args ... ;  ## generates callgrind.out.<pid>
```

Examples of build and run cycle:

```
## build app callgrind-ready
CXXFLAGS='-v -Og -ggdb3 -std=c++11 -march=sandybridge -mtune=sandybridge -DUSE_UNSTABLE_GEOS_CPP_API'

gcc ${CXXFLAGS} \
  -I/usr/include/c++/8 -I/usr/include/x86_64-linux-gnu/c++/8 \
  -I/usr/local/include/geos \
  RectangleIntersectsPerfTest.cpp  \
  -o dtest3 \
  -lm -lstdc++ -lpthread -lgeos


## generate a callgrind.out file
valgrind --tool=callgrind  dtest3

kcachegrind callgrind.out.20227
```

KCachegrind is a desktop browser for data produced by profiling tools.
The GUI application opens `callgrind.out.<pid>` files directly, and is available in Ubuntu in Development (universe) ; depends on **kde-runtime** and **qt4+**

see `kcachegrind-0.7.4/doc`

gcc compiler-generated code coverage hooks ----------

```
--coverage   ## convenience option, also adds link options  OR
-fprofile-arcs     -ftest-coverage   ## minimal flags  (see gcc manual)
```

GCC-native **code coverage** measurement.
Build an executable with `CXXFLAGS -coverage` , run once; coverage data file(s) are produced.
"When the compiled program exits it saves this data to a file called `auxname.gcda` for each source file"

Combine `.gcno` files generated at compile time, with `.gcda` files generated on first-run,
and function call behavior is completely recorded.

see gcc/Invoking-Gcov.html and gcc/Gcov-Data-Files.html

note that "coverage" is a seperate task from “profiling”

gcovr python ----------

Gcovr is a python utility for managing the use of the GNU `gcov` outputs;
generating both summarized and detailed code coverage measurement result reports.

Local install of gcovr

```
$   pip install gcovr --user
```

example, given `geos-exp/` holding built `.gcno` `.gcda` files, `gcovr_html/` is an empty dir:

```
$   gcovr -r geos-exp -j 6  --html-details --output=gcovr_html/index.html
```

gi88/ is a CMake-based build setup in alpha stage, that auto-invokes `gcovr` on an app
no autotools support, and requires one 3rd party CMake module at time of writing

gcov ----------

gcov creates a data file, usually `sourcefile.gcda` which includes
how many times each line of a source file `sourcefile.c` *has executed*,
and *how much computing time each section of code uses*.

Note `gcov` depends on fragile, local pathnames ! And only on code compiled with GCC.
The `gcov` tool must be run from the same directory as gcc was run (see **gcc** manual pages)

examples:

```
geos_master_cmake/build/bin$
  gcov -o ../tests/unit/CMakeFiles/test_geos_unit.dir/geom/TriangleTest.cpp.gcno \
    ./test_geos_unit geos::geom::Triangle


geos_master_cmake/build/bin$
  gcov -m \
    -o ../tests/unit/CMakeFiles/test_geos_unit.dir/geom/TriangleTest.cpp.gcno \
    -o ../tests/unit/CMakeFiles/test_geos_unit.dir/triangulate/*gcno \
    -o ../src/CMakeFiles/geos.dir/geom/Triangle.cpp.gcno \
      ./test_geos_unit geos::geom::Triangle
```

gprof ----------

enable: add `CXXFLAGS -pg`

A Call Graph Execution Profiler from the early days of *nix; display call graph profile data.

Note If you wish to perform line-by-line profiling you should use the `gcov` tool instead of `gprof`

types of outputs available from `gprof`
- flat profile
- call graph (text)
- annotated source code

gprof produces an execution profile of C or {other} programs. As a test-app is executed, the effect of called routines is incorporated into the profile of every caller. The profile data is taken from the call graph profile file (`gmon.out` default) which is created by programs that are compiled with the `-pg` option of "gcc" or {other}. The `-pg` option also links in versions of the library routines that are compiled for profiling.

example:
```
gprof --flat-profile  dtest2  gmon.out       ## ..is moderately useful
```

Symbol Files ----------

canonical files supplied by **Debian** packaging builds, with code libraries (others?)
e.g. `geos-debian-3.5.2/debian/libgeos-3.5.0.symbols`

https://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-symbols
https://www.debian.org/doc/manuals/maint-guide/advanced.en.html#librarysymbols

simple script to parse and emit CSV
```
parse_symbol.py   ## see repo TBD
```

manual hacks to get similar data include:
```
    nm -C libgeos.so > file.txt
   gdb info functions  > file.txt  ## break on main(), opt. load libgeos
   parse_nm.py     ## see repo TBD
```

hand parsing gcov tool session output can be made into a table like this:

```
perc_cov  loc  fname   gcov_fname
0   119  gi88/geos-exp/geos/src/algorithm/RobustDeterminant.cpp  RobustDeterminant.cpp.gcov
0   11   gi88/geos-exp/geos/src/index/sweepline/SweepLineInterval.cpp    SweepLineInterval.cpp.gcov
```
...
which can then be compared between builds, in SQL [SQL_A]

CLion + perf ----------

must run on bare metal, no VMs
CLion is commercial w/ 30-day trial license
https://www.jetbrains.com/help/clion/cpu-profiler.html

perf tool is an OS install
on Debian use `linux-tools-xxxx` like `linux-tools-4.15.0-43-generic`

```
sudo sh -c 'echo 1 > /proc/sys/kernel/perf_event_paranoid'
```

oprofile ----------

packaged in Debian; includes `operf` profiler tool
system-wide profiler for Linux systems

http://oprofile.sourceforge.net/doc/controlling-profiler.html

OProfile provides a low-overhead profiler (operf) capable of both single-application profiling and system-wide profiling. There is also a simple event counting tool (ocount).

GNU ISO C++ Profiler ----------

enable: #include profiler.h , `CXXFLAGS -D_GLIBCXX_PROFILE` # and variations

"zillions" of options
can write out a `libstdcxx-profile txt` file
requires code changes

GCC Instrumentation Directly to Functions ----------

enable: `CXXFLAGS -finstrument-functions`

Generate instrumentation calls for entry and exit to functions.
A manual alternative to `-pg` ; uses C hooks, described here:

balau82.wordpress.com/2010/10/06/trace-and-profile-function-calls-with-gcc

```
__cyg_profile_func_enter()  ;   __cyg_profile_func_exit()
// example in   test_geoms8/trace.c TBD
```

Profile Guided Optimization (PGO) ----------

see wikipedia entry

First Step:
`CXXFLAGS -fprofile-generate`

Enable options usually used for instrumenting application to produce profile useful for later recompilation with profile feedback based optimization.
You must use `-fprofile-generate` both when **compiling** and when **linking** your program.

The following options are enabled: `-fprofile-arcs, -fprofile-values, -fvpt.`

Second Step:
run the application built, notice the `.gcda` files (and others based on options)

To optimize the program based on the collected `.gcda` profile information,
rebuild the app with new option `-fprofile-use`.

Third Step:
`CXXFLAGS -fprofile-use`
Enable profile feedback-directed optimizations, and the following optimizations which are generally profitable only with profile feedback available: `-fbranch-probabilities, -fvpt, -funroll-loops, -fpeel-loops, -ftracer, -ftree-vectorize`, and `ftree-loop-distribute-patterns`.

see
StackOverflow for How to Use Profile Guided Optimizations in GCC
CProgramming.com 111902-pgo-amazing.html

[SQL_A]

```
##-------
##--  hack hack --  below is SQL after ingesting gcov text outputs, unstable

--
-- SQL hacks
-- make a table of post-processed gcovr output with python and hand-edit
--   test lines in groups of three; remove the 'no code to execute' lines
--   process into rows of text with python; output is script_out_parsed.tsv
--

CREATE TABLE public.parsed_gcov
(
  perc_cov double precision,
  loc integer,
  fname text,
  gcov_fname text
)

CREATE TABLE public.srcs_list ( fname text );

--
$ find geos-exp -name 'h$' > all_cpp_h.txt;
$ find geos-exp -name 'cpp$' >> all_cpp_h.txt
user=# copy srcs_list from '/home/user/srcs_live105/investigate_geos/gi88/all_cpp_h.txt';

--
select * from srcs_list ;
select count(*) from parsed_gcov ;
select count(*) from parsed_gcov where perc_cov > 80.0;
select count(*) from parsed_gcov where perc_cov < 50.0;

select b.fname from parsed_gcov a, srcs_list b where b.fname not in a.fname;

select b.fname from parsed_gcov a, srcs_list b where b.fname not in (select fname from srcs_list) ;

select a.fname , b.perc_cov, b.loc from srcs_list a LEFT JOIN parsed_gcov b on (a.fname = b.fname);
select a.fname , b.perc_cov, b.loc from srcs_list a LEFT JOIN parsed_gcov b on (a.fname = b.fname) order by
b.loc desc;

select * from parsed_gcov where fname ~ 'Geometry.cpp';

select fname from parsed_gcov where fname ~ 'h$';
select fname from parsed_gcov where fname ~ 'h$' order by fname;
select fname from parsed_gcov where fname ~ 'h$' and fname ~ 'usr' order by fname;

select count(fname) from srcs_list where fname ~ 'h$';
select a.fname , b.perc_cov, b.loc from srcs_list a LEFT JOIN parsed_gcov b on (a.fname = b.fname) order by
b.loc desc;
select a.fname , b.perc_cov, b.loc from srcs_list a LEFT JOIN parsed_gcov b on (a.fname = b.fname) where
b.loc is null order by b.loc desc;
select perc_cov, loc, fname from parsed_gcov order by loc desc;
select perc_cov, loc, fname from parsed_gcov order by loc desc;
select perc_cov, loc, fname from parsed_gcov where fname ~ 'Test' order by loc desc;
select perc_cov, loc, fname from parsed_gcov where fname !~ 'Test' order by loc desc;
```