Table of Contents

Code Analysis Tools

This page contain a guide to using various code profiling and analysis tools on the LHCb software. Please feel free to add any information to this page.

LHCb dedicated Profiling and Regression test service

LHCbPR is LHCb's own project to track performance issues in the general framework and the projects. Its aim is to help developers to obtain performing code.

Simplifying the problem, with GaudiDiff and GaudiExcise

The first step in debugging is often to provide the simplest reproducible example of the problem you've encountered. Tools are available to help you with that.

- To run only one algorithm from within a complicated sequence, take a look at GaudiExcise
- To see what the differences really are between to gaudi jobs, take a look at GaudiDiff

The "valgrind, callgrind and kcachegrind" Utilities

Valgrind is a general purpose utility for analyzing software. It contains various "tools" that perform tasks such as memory allocation checking, heap analysis and code profiling.

Building valgrind and callgrind

valgrind and callgrind can be built easily using the tar files available from the respective web pages (see above). However, it appears that the size of our LHCb applications is larger than can be handled by the default values coded into valgrind. This is seen by the error

```
--15591:0:aspacem  Valgrind: FATAL: VG_N_SEGMENTS is too low.
--15591:0:aspacem    Increase it and rebuild.  Exiting now.
```

It can be easily fixed, by doing as it says ! The file is coregrind/m_aspacemgr/aspacemgr-linux.c in the valgrind build directory (or coregrind/m_aspacemgr/aspacemgr.c in older releases). I found increasing the value from 2000 to 25000 seems to do the trick in all cases I've found so far. I also increased VG_N_SEGNAMES to 5000 from 1000.

Update : As per valgrind 3.9.0, this step is no longer required, as the default values have been increased to more reasonable settings.

Usage at CERN

System Valgrind

Check if you have "/usr/bin/valgrind" on your machine If not, on a desktop, it can be installed with: "sudo yum install valgrind"

Alternative Installation on machines with AFS

The version of valgrind installed by default on lxplus has problems running with applications that require reasonably large amounts of memory due to bugs in its internal memory model. It also cannot run on the normal lxplus nodes due to the virtual memory limit in place - e.g. you will see

```
[lxplus066] ~ > valgrind --version
valgrind: mmap(0x8bf5000, -1488932864) failed during startup.
valgrind: is there a hard virtual memory limit set?
```

These bugs have been fixed in the more recent versions than that installed. Unfortunately such a version is not available by default. A patched version is available in the LCG AFS area, which the following scripts provide access to :-

```
> source /afs/cern.ch/lhcb/group/rich/vol4/jonrob/scripts/new-valgrind.csh
```

or

```
> source /afs/cern.ch/lhcb/group/rich/vol4/jonrob/scripts/new-valgrind.sh
```

(for csh or bash like shells respectively) which currently provides the latest versions of valgrind and callgrind, with a private patch applied to allow valgrind to properly run with the LHC LCG applications.

Working with gaudirun.py

Unfortunately, valgrind cannot be used directly with `gaudirun.py`. There are three solutions, first you can use `gaudirun.py` to generate "old style" Job Options:

```
> gaudirun.py -n -v -o options.opts options.py
```

Depending on your options and the version of `gaudirun.py` used to generate them, `options.opts` may contain some lines which are not valid... If this happens simply edit the file by hand and remove these lines.

Then invoke valgrind with something like :-

```
> valgrind --tool=memcheck Gaudi.exe options.opts
```

Second, and probably better since it avoids the additional step of creating old style depreciated options (which does not always work), you can run valgrind directly on the `python` executable and pass the full path to `gaudirun.py` as an argument.

```
> valgrind --tool=memcheck python `which gaudirun.py` options.py
```

This second method will be used in the examples below.

Since Gaudi v23r7, a new way to invoke profilers has been introduced in Gaudi: the new --profilerName option allows to directly call valgrind (or igprof) on the Gaudi process. It is also possible to pass options directly to valgrind, with --profileExtraOptions.

NB1: The profiler name is valgrind+tool (e.g. "valgrindmemcheck" or "valgrindmassif" or "valgrindcallgrind"), see the `gaudirun.py --help` for more details.

NB2: In the profileExtraOptions "--" have to be replaced by "__" to avoid issues with the option parser.

NB3: valgrind needs to be in the path for this option to run.

```
gaudirun.py --profilerName=valgrindmemcheck --profilerExtraOptions="-v __leak-check=yes  __leak-c
```

Working with ROOT6

ROOT6 uses clang internally as its interpreter, and this causes valgrind some problems due to its use of 'self modifying code'. To deal with this the additional valgrind option `--smc-check` should be used. Add

`--smc-check=all-non-file`

To your set of command line options. This unfortunately makes valgrind much slower, but at least it works...

There is prehaps some hope that in the future when ROOT enables full support for clang's pre-compiled-modules, the need for this might go away...

Memory Tests

The main use of valgrind is to perform memory tests of your code for things like memory corruptions, memory leaks and uninitialised variables, using the `memcheck` valgrind tool. Full documentation of this tool is available in section 4 of the valgrind user guide.

It is useful to add a few additional options, to improve the quality of the output. The options

```
-v --error-limit=no --leak-check=yes --num-callers=50 --leak-check=full --track-origins=yes
```

increase the amount of information valgrind supplies.

In additional, there are some warnings which are well known about (for instance from the Gaudi framework or even thrid party libraries, like ROOT, STL or boost) and it is best to suppress these, otherwise the amount of output to deal with is huge. This can be done with `--suppressions=$ROOTSYS/etc/valgrind-root.supp --suppressions=$STDOPTS/valgrind-python.supp --suppressions=$STDOPTS/Gaudi.supp`.

To suppress some general warnings from the LHCb stack, not specific to a particular application, include `--suppressions=$STDOPTS/LHCb.supp`.

To generate in the output log template suppression blocks for new warnings, add the option `--gen-suppressions=all` to the command line arguments.

DaVinci has the specific suppression file `$DAVINCIROOT/job/DaVinci.supp`, so the final full command for DaVinci is then :-

```
> valgrind --tool=memcheck -v --error-limit=no --leak-check=yes --num-callers=50 --leak-check=fu
```

For Brunel, there is an additional suppressions file you can apply `$BRUNELROOT/job/Brunel.supp` giving :-

```
> valgrind --tool=memcheck -v --error-limit=no --leak-check=yes --num-callers=50 --leak-check=fu
```

The above commands will send the output to the terminal. It is best to redirect this to file. One additional complication is the valgrind output is sent to `STDERR`, not `STDOUT`, so you will need to redirect both to file... This can be done (for bash like shells) by appending `&>mem.log` to the full valgrind command (for csh like shells use `>& mem.log`).

Another useful trick is to first redirect `STDERR` to `STDOUT`, then use `tee` to send the output to file **and** to the terminal. For bash this is `2>&1 | tee profile.log`. For (t)csh this is `| & tee -a profile.log`.

This will produce a large output file contain detailed information on any memory problems. You can either simply read this file directly, or if you prefer use the alleyoop application to help interpret the errors.

If possible use a debug build (see below) as that will provide more information. This can be done by running

```
> LbLogin -c $CMTDEB
```

before any `SetupProject` calls.

Cache Profiling

The valgrind tool called "cachegrind" provides a simulation of the CPU caches and thus is a cache and branch-prediction profiler. In the simpliest case it can be activated with

```
> valgrind --tool=cachegrind application
```

See the cachegrind manual for more details.

Code Profiling

A valgrind tool called "callgrind" also exists which does some detailed code profiling. For valgrind versions 3.1.1 and earlier callgrind is not included in the main valgrind package, it must be installed seperately. As of valgrind 3.2.0, callgrind was integrated into the mainstream valgrind package. Full documentation of this tool is available in section 6 of the valgrind user guide.

In addition, a nice GUI is available to view the output of this tool, called "kcachegrind" (kcachegrind *can* view the output of cachegrind, but despite its confusing name it is actually primarily designed for callgrind).

See here for more details on callgrind and kcachegrind.

In the simple case, usage is just

```
> valgrind --tool=callgrind application
```

and where *application* is for example "Boole.exe options-file". This will produce an output file of the form callgrind.xxxxx, which can be read in by kcachegrind. More information on available command line options are given on the web page, or by running "callgrind --help or valgrind --help".

The options `--dump-instr=yes --trace-jump=yes` are also useful as they provide more information.

If you wish to include in the callgrind output the same cache profiling information as provided by cachegrind, include as well the options `--cache-sim=yes --branch-sim=yes`

One very useful option is "--dump-before" which can be used for the creation of an output file before calling particular methods. Using this with for instance BooleInit::execute allows the creation of one dump per event, which can then be read in individually. I.e. a full command line could be, for Boole

```
> valgrind --tool=callgrind -v --cache-sim=yes --branch-sim=yes --dump-instr=yes --trace-jump=ye
```

or for Brunel

```
> valgrind --tool=callgrind -v --cache-sim=yes --branch-sim=yes --dump-instr=yes --trace-jump=ye
```

and finally for DaVinci

```
> valgrind --tool=callgrind -v --cache-sim=yes --branch-sim=yes --dump-instr=yes --trace-jump=ye
```

Alternatively, it is sometimes more useful to average the profiling information over several events in order to get a better overall picture. This can be done using the following options, which will produce one dump for the `initialize()` phase, one for `execute()` and a third for `finalize()`.

```
> valgrind --tool=callgrind -v --cache-sim=yes --branch-sim=yes --dump-instr=yes --trace-jump=ye
```

Also note, valgrind generally requires a large amount of memory and CPU, and thus you may run into problems with the CPU time and virtual memory size limits in place on the general purpose lxplus nodes. If this happens, try using a private afs box (e.g. pclhcbXX.cern.ch) if such a machine is available to you. If not, you could try running on one of the lxbuildXXX machines (XXX = 035 or 042, although we should avoid all running at the same time on the same node ...).

kcachegrind is now available on lxplus - see here for details. Note, sourcing the above setup file (/afs/cern.ch/lhcb/group/rich/vol4/jonrob/scripts/new-valgrind.csh) adds the appropriate path for kcachegrind to your PATH.

Using kcachegrind takes some getting used to. One of the first things to do is add your code location (e.g. ~/cmtuser) to the list of known locations (settings -> configure kcachegrind). The kcachegrind web page contains more hints on getting started.

Remote Control

Sometimes its useful to be able to start and stop profiling by hand. This can be done by first passing the option

```
--instr-atstart=no
```

to the valgrind command used to start the application. This will start the process running, but profiling will not start until you run (on the same machine, so start a second terminal) the command

```
> callgrind_control --instr=on
```

later on you can then stop the profiling when you wish, using

```
> callgrind_control --instr=off
```

Job Options Driven Profiling

Callgrind can also be configured from within job options, e.g. :

```
 def addProfile():
     from Configurables import CallgrindProfile
     p = CallgrindProfile('CallgrindProfile')
     p.StartFromEventN = 40
     p.StopAtEventN = 90
     p.DumpAtEventN = 90
     p.DumpName = 'CALLGRIND-OUT'
     GaudiSequencer('RecoTrSeq').Members.insert(0, p)
 appendPostConfigAction(addProfile)
```

Moore offers an option to enable profiling with callgrind:

```
from Moore import options
options.callgrind_profile = True
```

The properties of `CallgrindProfile` as in the function above can be set by:

```
from PyConf.application import make_callgrind_profile
with make_callgrind_profile.bind(start=40, stop=90, dump=90, dumpName='CALLGRIND-OUT'):
    # run your favourite code, e.g. a reconstruction
    run_reconstruction(option, standalone_hlt2_forward_reco)
```

or globally by `make_callgrind_profile.global_bind(start=40, stop=90, dump=90, dumpName='CALLGRIND-OUT')`.

Then to run use the following command line.

```
gaudirun.py -T --profilerName=valgrindcallgrind --profilerExtraOptions="__cache-sim=yes __branch-
```

Profiling specific lines of code

Callgrind can be told to only profile certain lines of code, for example within a function. The following steps are needed:

- Get the header files:

  ```
  local_valgrind.h, local_callgrind.h
  ```

  e.g. from

  ```
  /cvmfs/lhcb.cern.ch/lib/lhcb/GAUDI/GAUDI_v30r5/GaudiProfiling/src/component/valgrind/loca
  ```
- Add the following to your code:

  ```
  #include "local_callgrind.h"
  // ...
  CALLGRIND_START_INSTRUMENTATION;
  // some code
  CALLGRIND_STOP_INSTRUMENTATION;
  ```
- run with

  ```
  ../run gaudirun.py --profilerName=valgrindcallgrind --profilerExtraOptions="__smc-check=
  ```
- Do not use

  ```
  CallgrindProfile
  ```

  as this seems to interfere with it.

Memory Usage Monitoring

The valgrind tool called "massif" also exists which does some detialed memory usage monitoring. Full documentation of this tool is available in section 9 of the valgrind user guide.

Simple usage is just

```
> valgrind --tool=massif python `which gaudirun.py` options.py
```

Note that command line options exist to control the level of memory usage monitoring that is applied, whether the stack is monitored as well as the heap (by default not). See the user guide for more details. Useful additional default options are, for example

```
> valgrind --tool=massif -v --max-snapshots=1000 python `which gaudirun.py` options.py
```

Also note, to get the most out of this tool (as with all valgrind tools) it is best to run the debug software builds.

Understanding the output

There are a selection of applications that can be used to analysis valgrind's output. This link contains some information on these. Alleyoop and Valkyrie are installed as part of the setup above.

Valkyrie requires an XML output file. To save the output to an XML file, use the command line options `--child-silent-after-fork=yes --xml=yes --xml-file=memcheck.xml` as well as any others you wish

to use.

This link is a snapshot of the Atlas valgrind TWiki, that contains some useful information.

Debugging gaudirun.py on Linux with gdb

The easiest way to debug with `gdb` is to use the built-in `--gdb` flag of `gaudirun.py`

```
gaudirun.py --gdb yourOptions.py
```

This picks up `gdb` from LCG. (LCG 95 comes with verison 8.2.1, while older releases have 7.12.1)

Note: If possible use a debug build (or at least a build with debug symbols) as that will provide more information. This can be done by running, for example,

```
LbLogin -c x86_64-centos7-gcc8-dbg  # for a debug build
LbLogin -c x86_64-centos7-gcc8-opt+g  # for an optimized build with debug symbols
```

Running with a different gdb

Alternatively, `gaudirun.py` applications can be run through the `gdb` debugger using a similar trick as with valgrind, to call python directly and pass `gaudirun.py` as an argument. Just type

```
gdb --args python `which gaudirun.py` yourOptions.py
```

and then to run the application then simply type `run` at the gdb command line. (The option `--args` tells gdb to interpret any additional options after the executable name as arguments to that application, instead of the default which is to try and interpret them as core files...)

When it crashes, type `where` to get a traceback at that point.

Note: Currently, the default `gdb` on lxplus is too old to be useful for gcc 4.8 (or later) builds. You can either use your system gdb or use one from cvmfs with

```
export PATH=/cvmfs/lhcb.cern.ch/lib/contrib/gdb/7.11/x86_64-slc6-gcc49-opt/bin:$PATH
```

Attaching GDB to a running process

GDB can be used to debug an already running process, which can be useful for instance to investigate hung applications. Just set up a second terminal with the same software environment as the one you wish to debug. Running the DEBUG builds is highly advised, as then you will get line numbers in any traceback information.

Then, identify the process ID for the job you wish to investigate, e.g.

```
 > ps x | grep gaudirun
 4200 pts/7    S+     0:00 grep gaudirun
32081 pts/6    Rl+   46:05 /cvmfs/lhcb.cern.ch/lib/lcg/releases/LCG_68/Python/2.7.6/x86_64-slc6-g
```

So in this case its 32081. Then it is just a matter of running :-

```
 > gdb `which python` 32081
```

This will take a short while, loading libraries etc. When done you can then investigate, for instance :-

```
(gdb) where
#0  0x00007fd11e5c68ac in G4PhysicsOrderedFreeVector::FindBinLocation(double) const () at ../mana
```

```
#1  0x00007fd11e5cc6f4 in G4PhysicsVector::ComputeValue(double) () at ../management/src/G4Physics
#2  0x00007fd119e5a302 in RichG4Cerenkov::PostStepDoIt(G4Track const&, G4Step const&) ()
    at /afs/cern.ch/lhcb/software/releases/GEANT4/GEANT4_v95r2p7g2/InstallArea/include/G4PhysicsV
#3  0x00007fd11e588130 in G4SteppingManager::InvokePSDIP(unsigned long) () at ../src/G4SteppingMa
<snip>
```

Note that the process being debugged will be paused whilst GDB is attached. If you quit GDB, it will then continue running, and if you wish you can reattach again at a later stage.

Scripting GDB with python

With recent version of GDB, it is possible to script the behavior of the debugger. This is quite handy when a lot of repetitive tasks is requested in the debugger itself, for example when debugging multi-threaded applications. Since GDB 7.0 (which is present on lxplus), it is distributed with python modules interacting directly with the GDB internals. One way to check that GDB has the support for this is to do:

```
(gdb) python print 23
23
```

at the GDB prompt. This must not fail.

More information can be fount at the PythonGdb page of the GDB Wiki and in the GDB documentation on how to use it.

GDB in Emacs

An alternative method is to use `Emacs` to start a debug session.

```
 > emacs
 M-X gdb  (or Esc-X on many keyboards)
 gdb python
 (gdb) run `which gaudirun.py` yourOptions.py
```

At CERN `gdb python` may give you an error, if that is the case you should do

```
/afs/cern.ch/sw/lcg/contrib/gdb/7.6/x86_64-slc6-gcc48-opt/bin/gdb python
```

You can use the emacs toolbar to set break in lines, unset them and issue debugger commands, or you can pass them as command lines at the **`(gdb)`** prompt. In which case here are a couple of useful short-cuts:

- **`(gdb) Ctrl- up-arrow/down-arrow`** allows to navigate through the commands you already typed

Google PerfTools

Google provides a set of performance tools. For details on usage within LHCb see here.

CPU-assisted performance analysis

All the previously discussed performance analysis tools are often unable to provide a precise quantitative analysis of what happens as a program is executed on a real CPU, for different reasons:

- Valgrind essentially works by emulating the execution of the program on a virtual CPU. This artificially inflates the cost of CPU computations with respect to other operations (such as I/O) by more than an order of magnitude, and entails that performance analysis must be based on a mathematical model of a CPU, which is in practice quite far off from what modern Intel CPUs

actually do.
- Google's profiler, like other user-space sampling profilers (gprof, igprof...), is only able to tell where a program spends its time, and not why. For example, it cannot tell where CPU cache misses are happening, which complicates memory layout optimizations.
- Neither of these tools is able to analyze the time spent in the operating system kernel, which is important for assessing the impact of blocking I/O operations or lock contention in multi-threaded code.

A more precise analysis of program execution on a real machine can be obtained from tools which leverage the Performance Monitoring Counters of modern CPUs, such as the "perf" profiler of the Linux kernel or Intel's VTune Amplifier. These tools provide an accurate and detailed picture of what is going on in the CPU as a program is executing, and have a negligible impact on the performance of the program under study when used correctly.

There is a price to pay for this precision, however, which is that the functionality provided by these tools depends on your system configuration. Some functionality may only be available on recent Intel CPUs, and other may require use of a recent enough Linux kernel.

Linux perf (also known as perf_events)

The perf profiler is a free and open source program which builds on the perf_events interface that has been integrated in the Linux kernel since Linux 2.6.31. It is highly recommended to use it with as recent a Linux kernel release as possible (at least 3.x) for the following reasons:

- Early 2.6.x versions had some very nasty bugs, causing system lock-up, for example.
- Due to the way it operates, perf requires CPU-specific support code. This means in particular that you are unlikely to be able to leverage your CPU's full performance monitoring capabilities if your Linux kernel version is older than your CPU model.
- Perf is evolving quickly, and new versions can also bring massive improvements in features, usability and performance.

You can learn more about the improvements brought by successive perf releases in the highlights and "Tracing/profiling" sections of the Linux kernel release notes at https://kernelnewbies.org/LinuxVersions , and check which Linux kernel version your system is running using the uname command:

```
> uname -r
4.14.1-1-default
```

To install the perf profiler, use your Linux distribution's package manager. The name of the package(s) to be installed varies from one distribution to another, here are some common ones:

- Debian/Ubuntu: linux-tools
- RedHat/CentOS/Fedora/SUSE: perf

The simplest thing which you can do with perf is to measure aggregated CPU statistics over the course of an entire program execution. This is done using the "perf stat" command:

```
> perf stat <your command>

[ ... normal program output ... ]

Performance counter stats for 'cargo run --release':

       4428.370578      task-clock (msec)         #    1.000 CPUs utilized
                46      context-switches          #    0.010 K/sec
                 0      cpu-migrations            #    0.000 K/sec
```

```
          6 459      page-faults               #    0.001 M/sec
   15 738 566 590      cycles                    #    3.554 GHz
   30 034 797 373      instructions              #    1.91  insn per cycle
    2 222 188 760      branches                  #  501.807 M/sec
       88 966 900      branch-misses             #    4.00% of all branches

      4.429156950 seconds time elapsed
```

The output of perf stat contains raw statistics (on the left) and some interpretations of the numbers (on the right). Here, we can see that the program under study is not multi-threaded (as the right column points out, only 1 CPU is utilized), but makes reasonably efficient use of the single CPU core that it runs on (at 1.91 instructions per cycle, we're not too far away from the theoretical Haswell maximum for this code). If a performance number is abnormally bad, perf will helpfully highlight it using color in your terminal (which we can't show in this wiki).

We can ask perf for more statistics using the "-d" command line switch:

```
> perf stat -d <your command>

[ ... normal program output ... ]

 Performance counter stats for 'cargo run --release':

      4425.711186      task-clock (msec)         #    0.999 CPUs utilized
             159      context-switches          #    0.036 K/sec
               0      cpu-migrations            #    0.000 K/sec
           6 431      page-faults               #    0.001 M/sec
   15 684 677 897      cycles                    #    3.544 GHz                      (50.10%)
   29 976 594 100      instructions              #    1.91  insn per cycle           (62.56%)
    2 208 236 648      branches                  #  498.956 M/sec                    (62.56%)
       88 851 174      branch-misses             #    4.02% of all branches          (62.60%)
    6 042 105 001      L1-dcache-loads           # 1365.228 M/sec                    (62.08%)
       11 320 634      L1-dcache-load-misses     #    0.19% of all L1-dcache hits    (25.06%)
        1 870 540      LLC-loads                 #    0.423 M/sec                    (25.02%)
          319 260      LLC-load-misses           #   17.07% of all LL-cache hits     (37.57%)

      4.431148695 seconds time elapsed
```

Here you can see that we start to get interesting information about the use of CPU caches. For this particular program, the cache usage pattern is that we rarely go out of the first-level CPU cache (L1), let alone all the way to the last-level cache (LLC), but that when we do reach for that one, we often need to go all the way to main memory. As main memory accesses are around 50x more costly than first-level cache accesses (the latency order of magnitudes being ~3 cpu cycles for L1 vs ~150 cycles for main memory), it is often useful to carefully examine both numbers. In this case they are ultimately not a concern: even when re-scaled with this order of magnitude in mind, our last-level cache misses still have negligible impact compared to the common case of L1 cache hits.

Another thing to pay attention to here is the new column of percentages on the right. CPU performance monitoring counters have some hardware limitations, the most important of which being that you can only monitor a small set of them at any given time. Here, because we are looking at a lot of different statistics at once, perf was forced to monitor only a subset of them at a time and constantly switch between them. It then interpolates the missing data samples, which has some overhead and reduces the quality of the measurement. The percentage tells you during which fraction of the total measurement time the corresponding performance counter was actually active.

If you know exactly which performance counters you are interested in, you can get more precise measurements by asking perf to only measure these ones, using the "-e" command line switch:

```
> perf stat -e L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses <your command>
```

```
[ ... normal program output ... ]

 Performance counter stats for 'cargo run --release':

     6 023 335 754      L1-dcache-loads                                               (74.90%)
         8 994 495      L1-dcache-load-misses     #    0.15% of all L1-dcache hits    (50.16%)
         1 407 348      LLC-loads                                                     (50.09%)
           311 839      LLC-load-misses           #   22.16% of all LL-cache hits     (75.00%)

       4.404909465 seconds time elapsed
```

As you can see, the L1-dcache-load-misses counter was now active 50% of the time, instead of 25% before, which means that we aggregated twice as many performance statistics over the same program run time. However, we lost the other performance counters. This is usually a good second step after a potential performance problem has been identified in the "generic" perf stat output.

You can get a full list of the (numerous) CPU performance counters supported by perf using the "perf list" command.

Perf stat is very powerful and has very low overhead, but it only gives you coarse-grained information. Often, you want to know where your program spends its time, and more importantly why. This information can be measured using the "perf record" and "perf report" commands. The first one analyzes the performance of your program by periodically sampling which function your code is executing and how the CPU performance counters are evolving, then it correlates these two pieces of information. The second one displays the resulting statistics in a nice text based user interface.

In order to report function names, your program must be compiled with debugging symbols (as enabled by the "-g" GCC flag, or the "Debug" and "RelWithDebInfo" CMake build configurations). To get a profile that is representative of your application's actual performance, you will obviously need to leave compiler optimizations on, which is exactly the kind of scenario that CMake's built-in "RelWithDebInfo" configuration was designed for.

When you run perf record for the first time, you will likely see a warning message like the following one:

```
> perf record <your command>
WARNING: Kernel address maps (/proc/{kallsyms,modules}) are restricted,
check /proc/sys/kernel/kptr_restrict.

Samples in kernel functions may not be resolved if a suitable vmlinux
file is not found in the buildid cache or in the vmlinux path.

Samples in kernel modules won't be resolved at all.

If some relocation was applied (e.g. kexec) symbols may be misresolved
even with a suitable vmlinux or kallsyms file.

Couldn't record kernel reference relocation symbol
Symbol resolution may be skewed if relocation was used (e.g. kexec).
Check /proc/kallsyms permission or run as root.

[ ... normal program output ... ]

[ perf record: Woken up 29 times to write data ]
[kernel.kallsyms] with build id d9c54397e4672f9850695351f23e25f24757f9b0 not found, continuing wi
[ perf record: Captured and wrote 7.910 MB perf.data (207305 samples) ]
```

This message warns you that perf is not currently allowed to report the names of the functions that you call within the Linux kernel. This ability can be very useful when the performance of your program is limited by system calls, and you want to understand what exactly is going on. If you have administrator rights on your machine, you can enable this feature by writing "0" in the /proc/sys/kernel/kptr_restrict pseudo-file. But we do

not need this feature for this short tutorial, and perf can live without it, so we'll do without for now.

So let's look at the report:

```
> perf report
Samples: 207K of event 'cycles:uppp', Event count (approx.): 106619947387
Overhead  Command          Shared Object              Symbol
  20.16%  07_io_bound_evl  libc-2.26.so               [.] _int_malloc
  14.66%  07_io_bound_evl  libc-2.26.so               [.] _int_free
  10.10%  07_io_bound_evl  07_io_bound_evloop.exe     [.] std::_Hashtable<int, int, std::allocato
   8.59%  07_io_bound_evl  07_io_bound_evloop.exe     [.] detail::ConditionSlotKnowledge::setupSl
   8.58%  07_io_bound_evl  07_io_bound_evloop.exe     [.] boost::detail::nullary_function<void ()
   6.85%  07_io_bound_evl  libc-2.26.so               [.] malloc
   3.89%  07_io_bound_evl  libc-2.26.so               [.] malloc_consolidate
   3.62%  07_io_bound_evl  07_io_bound_evloop.exe     [.] BenchmarkIOSvc::startConditionIO
   3.34%  07_io_bound_evl  libc-2.26.so               [.] cfree@GLIBC_2.2.5
   2.70%  07_io_bound_evl  libpthread-2.26.so         [.] __pthread_mutex_lock
   2.30%  07_io_bound_evl  libc-2.26.so               [.] tcache_put
   2.15%  07_io_bound_evl  07_io_bound_evloop.exe     [.] std::vector<detail::ReadySlotPromise, s
   2.05%  07_io_bound_evl  07_io_bound_evloop.exe     [.] std::_Sp_counted_base<(__gnu_cxx::_Lock
   1.91%  07_io_bound_evl  libc-2.26.so               [.] tcache_get
   1.87%  07_io_bound_evl  libstdc++.so.6.0.24        [.] operator new
   1.10%  07_io_bound_evl  libpthread-2.26.so         [.] __pthread_mutex_unlock_usercnt
[ ... shortened for brievity ... ]
```

This profile was acquired on a different program than the one which "perf stat" ran on at the beginning. This specific program could use more optimization work as it spends about half of its time in memory allocation related functions (malloc, free, and implementation details thereof), which is a common performance problem in idiomatic C++ code.

One piece of information which is noticable at the top of the table is that this profile was based on the "cycles" performance counter, which measures how many CPU clock cycles have elapsed. This is the most common performance indicator in early performance analysis, as it highlights where the program spends its time, which is what one is usually most interested in initially. However, one can use any performance counter here, using the same "-e" flag used with perf stat above. For example, "perf record -e L1-dcache-load-misses" would show which functions in your code are correlated with the most CPU cache misses.

There is one important piece of information which is missing from the above report, however, and that is the reason *why* some specific functions were called. When doing performance analysis, it is one thing to know that malloc() is called too often, but it is another to know why this happens. In this case, one wants to know who called these memory allocation functions, a piece of information also known as the call graph.

There are several ways to obtain a call graph, each with different advantages and drawbacks:

- The best method in almost every respect, when it can be used, is to use the Last Branch Record (LBR) hardware facility for this purpose. But this measurement method is only available on recent CPUs (>= Haswell for Intel), and there are hardware limitations on the depth of the call stacks that it can record.
- A universally compatible counterpart is to periodically make a copy of the program's stack and analyze it using the program's DWARF debug information. This is the same method used by the GDB debugger to generate stack traces. Sadly, the need to make stack copies gives this profiling method very bad performance, which means that perf can only measure the program's state rarely, and thus performance profiles must be acquired over much longer periods of time (several minutes) in order to be statistically significant. The profile files will also be much bigger, and slower to analyze.
- Sometimes, an alternative method based on sampling only the frame pointer of the program can achieve the same result at a much reduced cost, without loss of portability. Unfortunately, there is a very popular compiler performance optimization that breaks this profiling method, and even if you disable it on your code, the libraries that you use will most likely have it enabled. Therefore use of this profiling method is not recommended.

To measure a call graph, pass the "--call-graph=<method>" switch to perf record, where <method> will be either "lbr" or "dwarf" depending on which one your hardware allows you to use. Here is a DWARF-based version:

```
> perf record --call-graph=dwarf <your command> && perf report
[... an entire program execution later ... ]
Samples: 208K of event 'cycles:uppp', Event count (approx.): 105972566936
  Children      Self  Command          Shared Object               Symbol
+   46.21%     8.88%  07_io_bound_evl  07_io_bound_evloop.exe      [.] boost::detail::nullary_func
+   43.04%     0.02%  07_io_bound_evl  07_io_bound_evloop.exe      [.] boost::executors::basic_thr
+   41.85%     0.04%  07_io_bound_evl  07_io_bound_evloop.exe      [.] detail::SequentialScheduler
+   39.68%     8.32%  07_io_bound_evl  07_io_bound_evloop.exe      [.] detail::ConditionSlotKnowle
+   36.99%     1.90%  07_io_bound_evl  libstdc++.so.6.0.24         [.] operator new
+   32.67%     6.89%  07_io_bound_evl  libc-2.26.so                [.] malloc
+   26.47%    20.31%  07_io_bound_evl  libc-2.26.so                [.] _int_malloc
+   17.39%     9.89%  07_io_bound_evl  07_io_bound_evloop.exe      [.] std::_Hashtable<int, int, s
+   15.63%    14.44%  07_io_bound_evl  libc-2.26.so                [.] _int_free
+   14.21%     1.85%  07_io_bound_evl  07_io_bound_evloop.exe      [.] std::_Sp_counted_base<(__gn
+   10.21%     0.62%  07_io_bound_evl  libstdc++.so.6.0.24         [.] malloc@plt
+    6.11%     0.52%  07_io_bound_evl  07_io_bound_evloop.exe      [.] std::_Hashtable<int, int, s
+    5.76%     0.45%  07_io_bound_evl  07_io_bound_evloop.exe      [.] operator delete@plt
+    5.58%     0.01%  07_io_bound_evl  07_io_bound_evloop.exe      [.] boost::executors::executor_
+    5.54%     0.18%  07_io_bound_evl  07_io_bound_evloop.exe      [.] std::_Sp_counted_ptr_inplac
+    5.26%     0.03%  07_io_bound_evl  07_io_bound_evloop.exe      [.] boost::detail::shared_state
+    4.15%     3.97%  07_io_bound_evl  libc-2.26.so                [.] malloc_consolidate
+    4.02%     0.00%  07_io_bound_evl  07_io_bound_evloop.exe      [.] boost::detail::future_execu
+    3.99%     0.01%  07_io_bound_evl  07_io_bound_evloop.exe      [.] boost::detail::nullary_func
+    3.97%     0.04%  07_io_bound_evl  07_io_bound_evloop.exe      [.] ConditionSvc::setupConditio
+    3.90%     3.63%  07_io_bound_evl  07_io_bound_evloop.exe      [.] BenchmarkIOSvc::startCondit
+    3.43%     3.30%  07_io_bound_evl  libc-2.26.so                [.] cfree@GLIBC_2.2.5
+    3.34%     0.01%  07_io_bound_evl  07_io_bound_evloop.exe      [.] boost::detail::continuation
+    3.32%     0.00%  07_io_bound_evl  07_io_bound_evloop.exe      [.] benchmark
+    3.30%     0.00%  07_io_bound_evl  07_io_bound_evloop.exe      [.] main
[ ... shortened for brievity ... ]
```

There are two new things in the report:

1. The "Children" counter, which says which fraction of the elapsed CPU time was spent in a certain function *or one of the functions that it calls*. This allows one to tell, at a glance, which functions are responsible for the most of the execution time in your program, as opposed to which time was spent inside of each individual function. From a performance analysis perspective, this is a much more interesting piece of information than the previously displayed "self time" alone, which is why perf report automatically sorts functions according to this criterion when you enable call graph profiling.
2. The little "+" signs in the leftmost column of the report. These signs allow a recursive exploration of which functions a given function is calling, using the interactive perf report UI. For example, in the following text block, the places where "simulateEventLoop" function spends its time are expanded showing that a non-negligible time was spent inserting elements inside a hash table (itself part of a C++ unordered_set), which in turned caused a non-trivial fraction of dynamic memory allocations. Another time sink was the freeing of reference-counted data (from an std::shared_ptr), which caused expensive atomic operations and an eventual memory release.

```
Samples: 208K of event 'cycles:uppp', Event count (approx.): 105972566936
  Children      Self  Command          Shared Object               Symbol
+   46.21%     8.88%  07_io_bound_evl  07_io_bound_evloop.exe      [.] boost::detail::nullary_func
+   43.04%     0.02%  07_io_bound_evl  07_io_bound_evloop.exe      [.] boost::executors::basic_thr
-   41.85%     0.04%  07_io_bound_evl  07_io_bound_evloop.exe      [.] detail::SequentialScheduler
   - 41.80% detail::SequentialScheduler::simulateEventLoop
      - 37.31% detail::ConditionSlotKnowledge::setupSlot
         - 16.18% std::_Hashtable<int, int, std::allocator<int>, std::__detail::_Identity, std::e
            + 5.18% operator new
              1.03% operator new@plt
         + 13.28% std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>::_M_release
```

```
      + 3.85% boost::detail::shared_state_base::do_continuation
+   39.68%     8.32%  07_io_bound_evl  07_io_bound_evloop.exe     [.] detail::ConditionSlotKnowle
+   36.99%     1.90%  07_io_bound_evl  libstdc++.so.6.0.24        [.] operator new
+   32.67%     6.89%  07_io_bound_evl  libc-2.26.so               [.] malloc
+   26.47%    20.31%  07_io_bound_evl  libc-2.26.so               [.] _int_malloc
+   17.39%     9.89%  07_io_bound_evl  07_io_bound_evloop.exe     [.] std::_Hashtable<int, int, s
+   15.63%    14.44%  07_io_bound_evl  libc-2.26.so               [.] _int_free
+   14.21%     1.85%  07_io_bound_evl  07_io_bound_evloop.exe     [.] std::_Sp_counted_base<(__gn
+   10.21%     0.62%  07_io_bound_evl  libstdc++.so.6.0.24        [.] malloc@plt
```

This example also highlights one open issue with profiling C++ code, which is that the function names of C++ methods in idiomatic libraries such as the STL or boost can be gigantic, far away from the API names that are used in the code, and in general hard to read. Unfortunately, there is no good solution to this problem, the best that one can do is usually to look for interesting keywords in the long-winded C++ name (_Hashtable and _M_insert in the example above) and try to associate them with specific patterns in the corresponding function's code.

Note if you prefer to sort your call graph by caller rather than the (default) callee, use

```
perf report -g 'graph,0.5,caller'
```

This concludes this short introduction to perf. Here, we have only have scratched the surface of what perf can do. Other interesting topics could have included...

- Displaying annotated source code and assembly, in order to tell which part of a given function, exactly, takes time (bearing in mind that optimizing compilers can transform the source code of the original function hugely, which makes this analysis somewhat difficult).
- Measuring program activity every N-th occurence of a given event (e.g. L1 cache miss) instead of periodically, in order to more precisely pinpoint where in the code the event is occurring.
- The great many performance counters available on modern CPUs, which ones are most useful, and how their values should be interpreted.
- System-wide profiling, allowing one to study what happens to threads even when they "fall asleep" and call the operating system's kernel for the purpose of performing IO or locking a mutex.

...but this would go beyond the scope of this introductory TWiki page. For more detailed information on Linux perf, highly recommended sources of information and "cheat sheets" include the man pages of the various perf utilities and http://www.brendangregg.com/perf.html.

Intel VTune Amplifier

VTune uses the same performance analysis techniques as perf, but is commercially supported by Intel. This comes with different trade-offs: you must buy a very expensive (~1k$) license if you want to use it on your personal computer, but as long as you are able to rely on the licenses that are provided by CERN Openlab (or perhaps your institution), you will be able to enjoy a very nice and powerful graphical user interface, high-quality support and documentation from Intel, and periodical tutorials from Openlab. Obviously, you shouldn't expect it to work reliably on any CPU which has not been manufactured by Intel.

To use the tools, see the instructions at https://twiki.cern.ch/twiki/bin/view/Openlab/IntelTools.

A Gaudi auditor has been provided by Sascha Mazurov to interface to the VTune Intel profiler. See IntelProfiler, IntelProfilerExample and Video tutorial on profiler installation in Gaudi, running and analyzing it from command line (without GUI). Note that the Profiler is now part of the main Gaudi project, so it is no longer necessary to manually check it out from gitlab. It is not built by default though, so to enable it you must first setup the environment as per the instructions above and then build your checkout of the Gaudi project.

Memory Profiling with Jemalloc

Since Gaudi v26r3 it's possible to use Jemalloc profiling tools to audit memory allocations. Instructions can be found in Gaudi doxygen pages.

To view the diff between two memory dumps you can use the jeprof tool:

```
lb-run --ext jemalloc LCG/latest jeprof --evince --base=<first>.heap <executable> <second>.heap
```

Debug and Optimised Libraries

To build and run in debug mode, run

```
> LbLogin -c $CMTDEB
```

Before any `SetupProject` calls.

Utilities like callgrind, Google PerfTools and gdb can work on both optimised (`CMTCONFIG`) and un-optimised (`CMTDEB`) builds, although when running on optimised builds bear in mind you will not get all possible information. For instance inline functions, by their nature, are optimised away (those which are actually inlined by the compiler) and thus are not seen by the utility. Similarly you will not get annotated source code listings.

This does not happen in un-optimised (debug) builds, where all information is available, but of course you must then bear in mind that the two builds are different, and for some studies like profiling (code timing) the information you get will be different. In these cases a good approach is to first try running the normal optimised build. If you find you want finer grained information than that this method provides, then try the debug build.

Also, much more information is available if the "-g" option is used. Our un-optimised debug (CMTDEB) builds have this but not in our optimised (CMTCONFIG) builds. If you want to use this with your optimised builds, this can be done by checking out a private version of GaudiPolicy, update the optimisation flags in the requirements file to include "-g" as well as "-O2" and rebuild the libraries you wish to profile. Note this will increase the size of the binaries so if you have many libraries and a small AFS user quota you might need to increase it (email Joel Closier) or use another system without the strict quotas applied on AFS (like home institute systems).

Further reading

- Monir Hadji's talk on Tools for measuring code performance at 9th hackathon of software for the upgrade, 2017-12-12
- Monir Hadji's talk on measuring code performance with valgrind, perf and vtune at 9th Computing Workshop, 2017-05-18
- Stefan Nies' talk on tools for code optimisation at 37th software week, 2009-06-18
- Talks given at CHEP2009 and CHEP2010 giving hints on strategies and tools for optimising code:
    - Modular Software Performance Monitoring (Daniele Francesco Kruse and Karol Kruzelecki, LHCb-PROC-2010-070. Published in CHEP2010 proceedings)
    - CMS Software Performance Strategies
    - HEP C++ meets reality -- lessons and tips
- Similar talks were given at CERN on 2009-04-16 at a MultiCore R&D meeting dedicated to performance monitoring

---

ChrisRJones - 27 Feb 2006 MarcoCattaneo - 2017-12-12

This topic: LHCb > CodeAnalysisTools
Topic revision: r133 - 2020-01-15 - PaulAndreGunther1