Profiling tools for `LArSoft` code

Gianluca Petrillo

MicroBooNE Offline Software Retreat
Yale University, March 19th , 2014

What is “profiling”

Profiling:

the assessment of the resources that a program uses

- how much memory the program needs?
- how much disk space?
- how long it takes to run?

The optimization of a program starts from the analysis of the portions of it which take the largest resources:

1. find which part of code (ab)uses resources
2. understand *how* the resources are being used (and why)
3. act: fix the bug, try a different approach or redesign in full

`art` `Timing` service

What `Timing` service does

At the end of each `art` *module*, the time elapsed by it is printed out.
The time is based on the “wall clock”.
It also provides the total time for the event, and a summary in the end.

+ out of the box with `art`
+ usually FCL files already have it enabled
+ if not, just add `services.Timing: {}` to the configuration
– rough code identification: just tells the module

`art` `Timing` service in action

The following timing is part of a 10-event run with 3D reconstruction, (special configuration by Eric Church):

```
lar -c ./standard_reco_uboone_3D_cosmic_eric20140313.fcl \
-s prodgenie_bnb_nu_cosmic_3window_uboone_15367667_4_gen_15367673_4\
_g4_15367682_4_detsim_tpc_15367771_4_detsim_optical_15367859_4_reco2D.root
```

It runs quite a number of reconstruction modules (13), plus some.

The format is:

```
TimeModule> run: 1 subRun: 5 event: 50 trackkalmanhit Track3DKalmanHit 105
```

meaning that the event #50 of run #1, subrun #5 has spent 105" (wall clock time) running a `Track3DKalmanHit` module instance named `trackkalmanhit`.

art Timing service in action

This is well-formex text, which can be easily parsed. I have written my own (`SortModuleTimes.py LogFile`):

```
[basicstyle=\ttfamily]
RandomNumberSaver[rns]                          7.46965e-05" (RMS 20.3%)...
Track3DKalmanHit[trackkalmanhit]                84.5906"     (RMS 35.2%)...
Track3DKalmanHit[trackkalmanhitcc]              63.3171"     (RMS 38.2%)...
SpacePointFinder[spacepointfinder]              11.6491"     (RMS 54.3%)...
Track3DKalmanSPS[trackkalsps]                   3.79274"     (RMS 35.6%)...
BezierTrackerModule[beziertracker]              29.1193"     (RMS 29.3%)...
SpacePointFinder[spacepointfindercc]            5.33569"     (RMS 51.1%)...
BezierTrackerModule[beziertrackercc]            22.72"       (RMS 43.5%)...
Calorimetry[trackkalmanhitcalo]                 13.1713"     (RMS 37.1%)...
Calorimetry[trackkalmanhitcccalo]               9.71266"     (RMS 34.5%)...
Calorimetry[trackkalspscalo]                    7.75189"     (RMS 49.4%)...
BezierCalorimetry[beziertrackercccalo]          0.00275586"  (RMS 21.8%)...
BezierCalorimetry[beziertrackercalo]            0.0031462"   (RMS 18.1%)...
BeamFlashCompatibilityCheck[beamflashcompat]    0.00579774"  (RMS 20.2%)...
TriggerResultInserter[TriggerResults]           3.03507e-05" (RMS  9.6%)...
RootOutput[out1]                                0.599855"    (RMS 25.4%)...
=== events ===                                  252.207"     (RMS 34.9%)...
```

`Track3DKalmanHit` is our next stop...

Google `gperftools`

What `gperftools` does

`gperftools` provides statistical usage information for all the functions.
The job is sampled 100 times per second.

- \+ full call stack information reduces ambiguities
- \+ in-function timing
- \+ less than 2% overhead (i.e., slowdown)
- \+ conversion to `callgrind` format
- – sampling: information is subject to error
- – documentation is not very good; does it cope well with a busy system?
- – I couldn't actually extract the full stack information...

`gperftools` is currently not provided by UPS (but we could fix that).

gperftools: usage

gperftools requires their library to be linked to the program to be profiled. The easiest way is to do it dynamically:

```
env LD_PRELOAD=libprofiler.so CPUPROFILE=gperftools.prof \
lar -c ./standard_reco_uboone_3D_cosmic_eric20140313.fcl \
  -s prodgenie_bnb_nu_cosmic_3window_uboone_15367667_4_gen_15367673_4\
_g4_15367682_4_detsim_tpc_15367771_4_detsim_optical_15367859_4_reco2D.root
```

or

```
larrun.sh --gperf \
  -s prodgenie_bnb_nu_cosmic_3window_uboone_15367667_4_gen_15367673_4\
_g4_15367682_4_detsim_tpc_15367771_4_detsim_optical_15367859_4_reco2D.root
  ./standard_reco_uboone_3D_cosmic_eric20140313.fcl
```

(larrun.sh is a helper script, a seldom-updated version available in LArSoft wiki profiling page).
The output can be converted to "callgrind" format by

```
pprof -callgrind $(which lar) gperftools.prof > gperftools-callgrind.txt
```

Inclusive and exclusive timing

Profilers usually report two timings with complementary information.

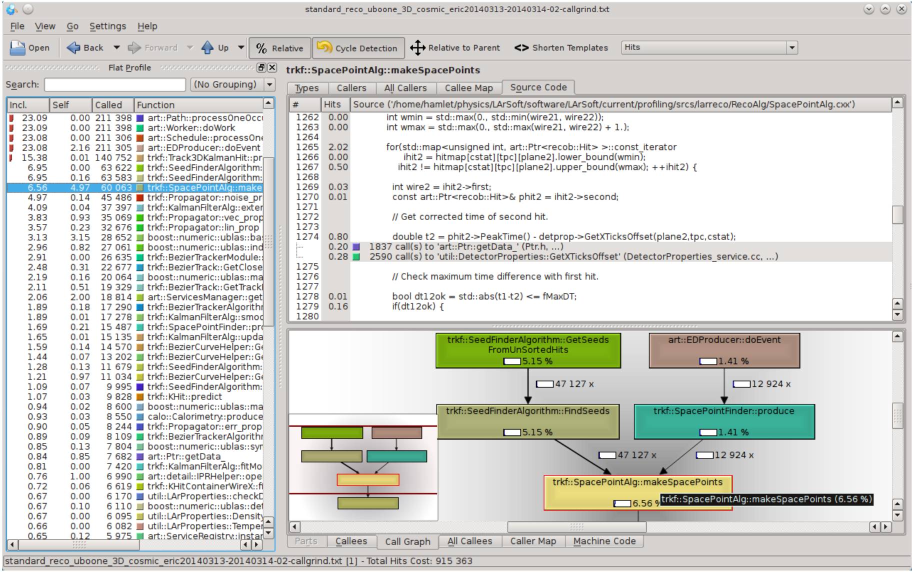
inclusive time
: is how much time the function takes from the beginning to the end, including *everything* it happens inside it

exclusive time
: is the time spent inside the function code, excluding the calls to other functions

```
void SortedInsertion(std::vector<double>& vect, double value) {
  size_t i = vect.size();
  while (i > 0) if (value > vect[--i]) {
    vect.insert(vect.begin()+i+1, value);
    return;
  } // while
  vect.insert(vect.begin(), value);
} // SortedInsertion()
```

The inclusive time includes all the time of the function, while the exclusive time will include the time spent in the loop and the check, but not the one in `std::vector::insert`, which is a separate function.

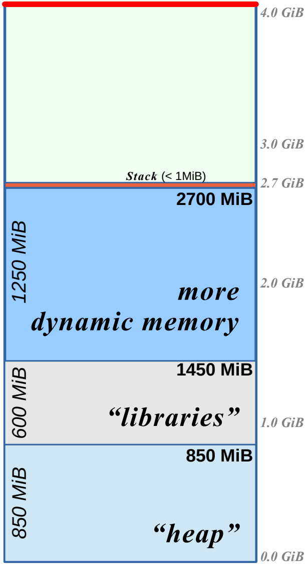
A visualizer for `callgrind`



The memory “footprint” of a program is how much memory the program needs in order to run.
We start to be interested to it when remote sites kill our job based on such a figure, for example 4 GB,. The memory is made of:

- **libraries** (mostly) not our fault, but it’s accounted on us nevertheless
- **stacks** where the local variables stay; usually negligible
- **heap & more** where the dynamic memory is allocated (via `new`)

The total of this memory is known as *virtual size* (VSIZE). The resident size (RSS) excludes the libraries (and something more).



art SimpleMemoryCheck service

What SimpleMemoryCheck service does

At the end of each art *module*, if the memory usage is increased, that's printed out.
The reported memory is both "VSIZE" and "RSS".
A summary at the end also shows peak memory usage.

This tool is good to detect steadily increase on memory and a first guess of where it happens.

+ out of the box with art
+ most of MicroBooNE FCL files already have it enabled
+ if not, just add `services.SimpleMemoryCheck: {}` to the configuration

– rough code identification: just tells the module
– rough memory estimation: only samples module boundaries
– reported peak memory can be far from the actual one
– sometimes the reported VSIZE is suspicious (smaller than RSS)

`massif` (`valgrind` tool)

What `massif` does

`massif` monitors *all* the memory allocations, and where they occur. It samples frequently, but reports details only when the memory changes dramatically.

+ precise information: how much memory allocated for each function in a call stack
+ possibility to track memory at lower level (`mmap` instead of `malloc`/`new`)
+ possibility to track stacks
+ graphic visualizers help

– detailed sampling can miss the part you are interested in
– `valgrind`-slow; tracking stacks is `valgrind`^{2}-slow

`massif` (`valgrind` tool): usage

Run it straight with:

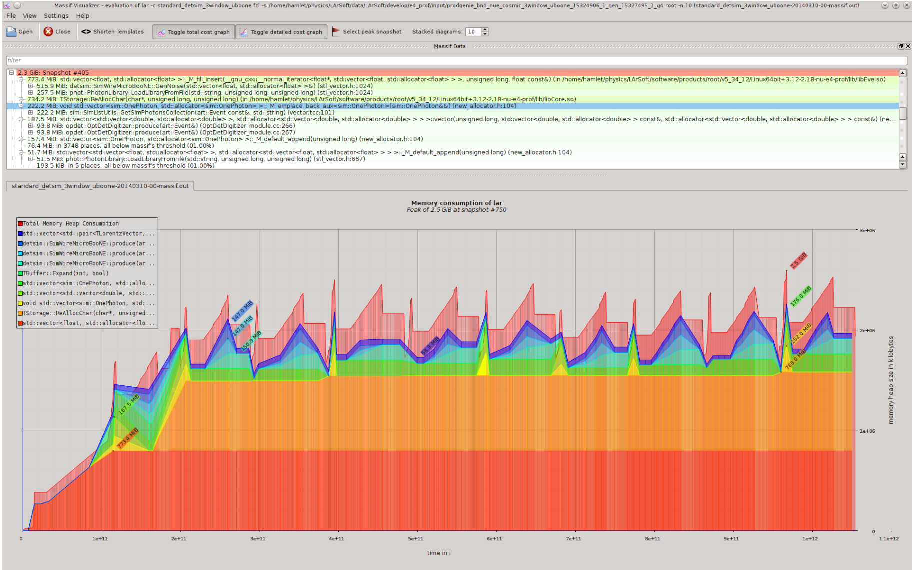
```
valgrind --tool=massif -n 5 \
lar -c ./standard_reco_uboone_3D_cosmic_eric20140313.fcl \
  -s prodgenie_bnb_nu_cosmic_3window_uboone_15367667_4_gen_15367673_4\
_g4_15367682_4_detsim_tpc_15367771_4_detsim_optical_15367859_4_reco2D.root
```

or

```
larrun.sh --massif \
  -s prodgenie_bnb_nu_cosmic_3window_uboone_15367667_4_gen_15367673_4\
_g4_15367682_4_detsim_tpc_15367771_4_detsim_optical_15367859_4_reco2D.root
  ./standard_reco_uboone_3D_cosmic_eric20140313.fcl -n 5
```

Then do something else. Chances are that you find the problem by looking at the code before `massif` is done.

massif example: general shape (detsim)



Profiling times

Currently, I can run with `e4:prof` code:

- plain run: 100-event samples (thousands should not be a problem)
- statistical CPU speed profiling: just a few percent overhead
- complete call profiling: 5-event chunks
- memory profiling: roughly as the complete call profiling (it's still `valgrind`)
- memory and stack profiling: 3-event chunks (takes longer than just memory)

A 8 GB memory machine would help making this quicker (virtual memory is deadly — for many people, if the machine is shared).

Stack profiling has shown to be not necessary (fortunately!).

Outlook

- each profiling tool has its strengths
- they must be understood, or wrong conclusions will be drawn
- best coding practises should be routinely applied (not “at the end”)
- testing is an important part of both the development and the optimization process

Best coding practises?

I am no coding authority, so my opinion follows:

- design and implement a test together with the code
- use dynamic allocation (`new` operator) only if a pointer to the data is needed
- never allow "naked" pointers to exit the creation scope
- when writing large-iteration loops, move the constant things out of them
- for the analysis of a lot of items, apply all the analysis steps to one item, rather than performing one step on all items
- beware of continuous reallocation of memory (`std::vector::push_back`), reserve the memory beforehand when possible
- reduce the copies of objects: use references, in-place construction (`std::vector::emplace_back`) or, if possible, `std::move`

Supplemental material

What is memory fragmentation

```
std::vector<TMatrixD> M;
std::vector<TVectorD> V;
for (size_t i = 0; i < BigNumber; ++i) {
   M.push_back(TMatrixD(36));
   V.push_back(TVectorD(36));
}
M.clear();
double* data = new double[BigBummer];
```

What is memory fragmentation

```
std::vector<TMatrixD> M;
std::vector<TVectorD> V;
for (size_t i = 0; i < BigNumber; ++i) {
   M.push_back(TMatrixD(36));
   V.push_back(TVectorD(36));
}
M.clear();
double* data = new double[BigBummer];
```

What is memory fragmentation

```
std::vector<TMatrixD> M;
std::vector<TVectorD> V;
for (size_t i = 0; i < BigNumber; ++i) {
   M.push_back(TMatrixD(36));
   V.push_back(TVectorD(36));
}
M.clear();
double* data = new double[BigBummer];
```

What is memory fragmentation

```
std::vector<TMatrixD> M;
std::vector<TVectorD> V;
for (size_t i = 0; i < BigNumber; ++i) {
   M.push_back(TMatrixD(36, 36));
   V.push_back(TVectorD(36));
}
M.clear();
double* data = new double[BigBummer];
```

What is memory fragmentation

```
std::vector<TMatrixD> M;
std::vector<TVectorD> V;
for (size_t i = 0; i < BigNumber; ++i) {
   M.push_back(TMatrixD(36));
   V.push_back(TVectorD(36));
}
M.clear();
double* data = new double[BigBummer];
```

The red rectangle outlines the heap memory usage of the program. Note the white holes of unused memory.

Why to use std::vector::reserve

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

We start with an empty vector.

Why to use std::vector::reserve

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

We push in the first element:

1. memory is allocated for the element

Why to use `std::vector::reserve`

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

We push in the first element:

1. memory is allocated for the element
2. the object is *copied*

Why to use std::vector::reserve

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

We push in the second element:

1. memory is allocated for two elements (in fact, `gcc`'s STL vector doubles the space)

Why to use `std::vector::reserve`

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

We push in the second element:

1. memory is allocated for two elements (in fact, `gcc`'s STL vector doubles the space)
2. the existing object is *moved*

Why to use std::vector::reserve

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

We push in the second element:

1. memory is allocated for two elements (in fact, gcc's STL vector doubles the space)
2. the existing object is *moved*
3. the old memory is freed; the new object is *copied*

Why to use `std::vector::reserve`

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

We push in the third element:

1. memory is allocated for four more elements (again doubling the space)

Why to use std::vector::reserve

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

We push in the third element:

1. memory is allocated for four more elements (again doubling the space)
2. the existing objects are *moved*

Why to use `std::vector::reserve`

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

We push in the third element:

1. memory is allocated for four more elements (again doubling the space)
2. the existing objects are *moved*
3. the old memory is freed; the new object is *copied*

Why to use `std::vector::reserve`

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

We had:

- 3 copy-constructed objects
- 3 memory allocations
- 2 memory deallocations
- 2 “fast” copies (*possibly* not involving copy constructors)
- some fragmented memory in the end

Had we used a `v.reserve(4)` call after defining the vector:

- 3 copy-constructed objects
- 1 memory allocation
- compact memory in the end

CPU profiling: options

We have different possibilities to collect that information:

inclusive timing
: based on the system clock, it can be the overall run time (e.g. `time` builtin in `bash`) or time report at checkpoints

sampling
: peek into the program at random times to see what it is doing (same principle as Monte Carlo integration)

stepping
: monitor every function call and how long it takes to complete

CPU profiling tools

`time` (`bash` command): per-run information

`Timing` (`art` service): per-event, module-level information

`gperftools` (Google) quick snapshot of where time is spent with full call history

`callgrind` (`valgrind` tool) count of each call and used cycles

A number of other tools are available: GNU `gprof` (told not to deal too well with C++11 yet), FermiLab FAST (same features as `gperftools`, but falling behind as the PC architecture evolves), Open|SpeedShop (also same features as `gperftools`, plus a nice interface, but a nightmare to compile), `IgProf` (I haven't tried), ...

art Timing service in action

(by `grep ^Time LogFile | uniq`)

```
TimeModule> run: 1 subRun: 5 event: 50 rns RandomNumberSaver 6.91414e-05
TimeModule> run: 1 subRun: 5 event: 50 trackkalmanhit Track3DKalmanHit 105.286
TimeModule> run: 1 subRun: 5 event: 50 trackkalmanhitcc Track3DKalmanHit 87.4242
TimeModule> run: 1 subRun: 5 event: 50 spacepointfinder SpacePointFinder 21.3958
TimeModule> run: 1 subRun: 5 event: 50 trackkalsps Track3DKalmanSPS 3.35858
TimeModule> run: 1 subRun: 5 event: 50 beziertracker BezierTrackerModule 39.4198
TimeModule> run: 1 subRun: 5 event: 50 spacepointfindercc SpacePointFinder 7.3756
TimeModule> run: 1 subRun: 5 event: 50 beziertrackercc BezierTrackerModule 26.9663
TimeModule> run: 1 subRun: 5 event: 50 trackkalmanhitcalo Calorimetry 14.1294
TimeModule> run: 1 subRun: 5 event: 50 trackkalmanhitcccalo Calorimetry 8.17911
TimeModule> run: 1 subRun: 5 event: 50 trackkalspscalo Calorimetry 6.37939
TimeModule> run: 1 subRun: 5 event: 50 beziertrackercccalo BezierCalorimetry 0.00312591
TimeModule> run: 1 subRun: 5 event: 50 beziertrackercalo BezierCalorimetry 0.00385594
TimeModule> run: 1 subRun: 5 event: 50 beamflashcompat BeamFlashCompatibilityCheck 0.00662398
TimeModule> run: 1 subRun: 5 event: 50 TriggerResults TriggerResultInserter 3.09944e-05
TimeModule> run: 1 subRun: 5 event: 50 out1 RootOutput 0.583466
TimeEvent> run: 1 subRun: 5 event: 50 320.972
TimeReport ---------- Time  Summary ---[sec]----
TimeReport CPU = 2517.729662 Real = 2542.718000
TimeReport> Time report complete in 2529.08 seconds
```

gperftools in action

The output of `gperftools` is binary, and it can be converted in text or other formats by `pprof` script (provided with `gperftools`):

```
pprof --text lar gperftools.prof > gperftools.txt
```

The top of the output:

```
23681    9.3%   9.3%    23681    9.3% boost::numeric::ublas::basic_row_major
17295    6.8%  16.1%    17295    6.8% __log10_finite ??:?
16625    6.5%  22.6%    16625    6.5% std::local_Rb_tree_increment tree.cc:?
11845    4.6%  27.2%    11845    4.6% __exp_finite ??:?
 9677    3.8%  31.0%     9677    3.8% _IO_str_seekoff ??:?
 9462    3.7%  34.7%     9462    3.7% trkf::SpacePointAlg::makeSpacePoints@8
 6720    2.6%  37.3%    11757    4.6% trkf::SpacePointAlg::makeSpacePoints@8
```

We can see the number of samples we collected in each function, the fraction respect to the total, “exclusive” timing (and its cumulative value), the samples we collected in the functions or in the ones it called, and its fraction, “inclusive” timing.
Let’s sort after the latter (`-cum` option):

gperftools listing

The top of the output, skipping `main`-like functions:

```
    0   0.0%   0.0%   211908  83.0% main
[...]
    0   0.0%   0.0%   140808  55.2% trkf::Track3DKalmanHit::produce
    0   0.0%   0.0%    63622  24.9% trkf::SeedFinderAlgorithm::GetSeedsFro
  513   0.2%   0.2%    63584  24.9% trkf::SeedFinderAlgorithm::FindSeeds
19659   7.7%   7.9%    61726  24.2% trkf::SpacePointAlg::makeSpacePoints@8
   13   0.0%   7.9%    50961  20.0% matrix_assign (inline)
    8   0.0%   7.9%    45492  17.8% trkf::Propagator::noise_prop
   99   0.0%   7.9%    37397  14.6% trkf::KalmanFilterAlg::extendTrack
  155   0.1%   8.0%    36997  14.5% trkf::Propagator::vec_prop
   16   0.0%   8.0%    32677  12.8% trkf::Propagator::lin_prop
  572   0.2%   8.2%    30938  12.1% boost::numeric::ublas::indexing_matrix
[...]
    0   0.0%  19.9%    26636  10.4% trkf::BezierTrackerModule::produce
    4   0.0%  19.9%    26163  10.2% ServiceHandle (inline)
 2258   0.9%  20.7%    25812  10.1% apply (inline)
   12   0.0%  20.7%    25130   9.8% calo::Calorimetry::produce
  464   0.2%  20.9%    24273   9.5% boost::numeric::ublas::matrix_assign
  854   0.3%  21.3%    23172   9.1% trkf::BezierTrack::GetClosestApproach
[...]
    1   0.0%  30.2%    17290   6.8% trkf::BezierTrackerAlgorithm::FilterOv
   29   0.0%  30.2%    17287   6.8% trkf::KalmanFilterAlg::smoothTrack
```

A visualizer for `callgrind`

`gperftools` has a lot more of information, but it's not easy to get to it.

```
pprof --callgrind lar gperftools.prof > gperftools-callgrind.txt
```

converts the information in `callgrind`'s format. Why do we care?

There are some graphic visualizer for callgrind!

- for KDE: `kcachegrind` (OK, it's for `cachegrind`... same format)
- in Eclipse's Linux Tools Project

Beware!!!

The conversion is far from being perfect, some values are wrong!

`valgrind`

`valgrind` is *a sort of CPU emulator*, which dresses ("instruments") each instruction that the program runs.
The monitoring of the program is almost complete.
The price is an almost unbearable execution overhead.
On top of `valgrind`, a number of "tools" are available:

- `memcheck` memory leak detection (the "default" tool)
- `callgrind`, `cachegrind` CPU profiling
- `massif`, `dhat` memory profiling
- `nulgrind` "will run roughly 5 times more slowly than normal, for no useful effect"
- ... and a few more

`callgrind` (`valgrind` tool)

What `callgrind` does

`callgrind` monitors *all* the function calls, counting how many times they happen and how long they take.

- \+ precise information: how many times each function has been called, how many CPU cycles it has used
- \+ in-function timing
- – information can be misleading (there no full stack information, the call graph is less informative than it looks)
- – `valgrind`-slow

Run it straight with:

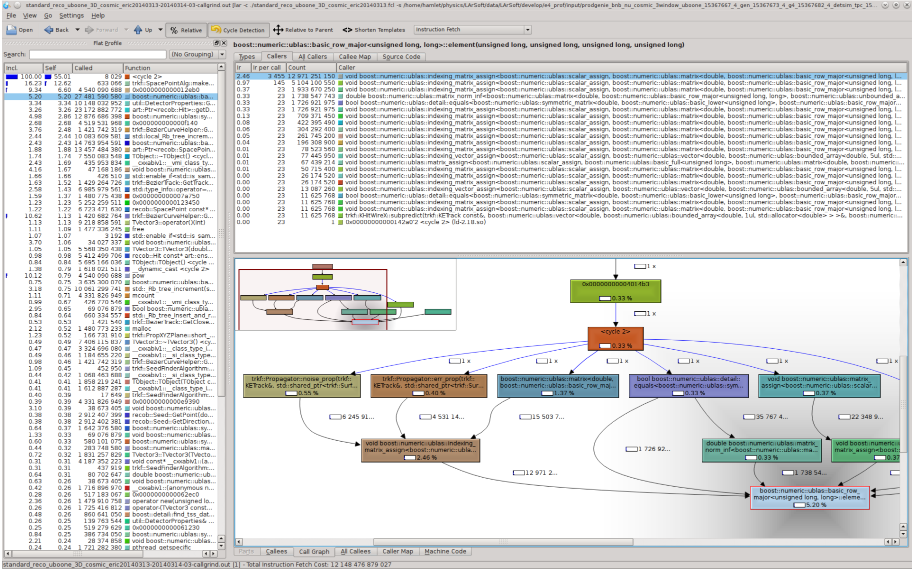
```
valgrind --tool=callgrind -n 5 \
lar -c ./standard_reco_uboone_3D_cosmic_eric20140313.fcl \
  -s prodgenie_bnb_nu_cosmic_3window_uboone_15367667_4_gen_15367673_4\
_g4_15367682_4_detsim_tpc_15367771_4_detsim_optical_15367859_4_reco2D.root
```

or

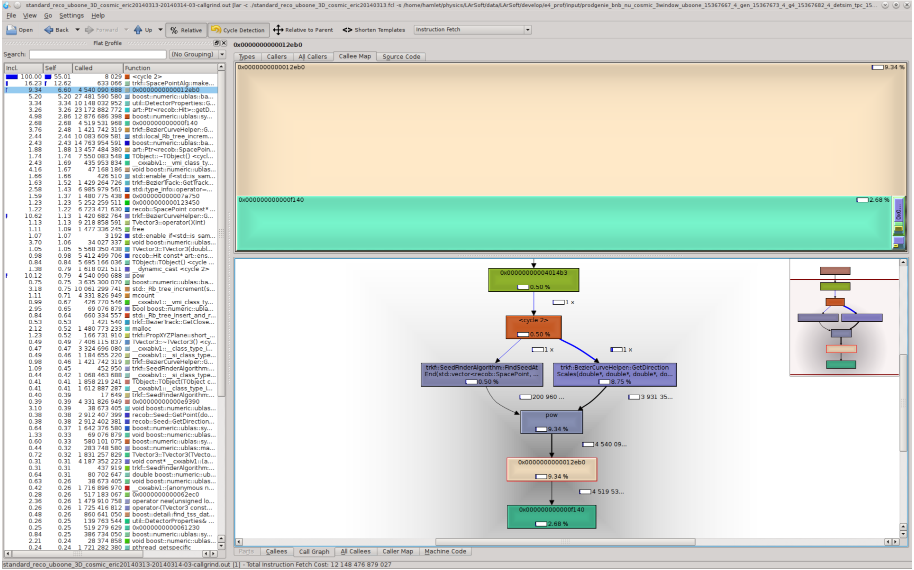
```
larrun.sh --callgrind \
  -s prodgenie_bnb_nu_cosmic_3window_uboone_15367667_4_gen_15367673_4\
_g4_15367682_4_detsim_tpc_15367771_4_detsim_optical_15367859_4_reco2D.root
  ./standard_reco_uboone_3D_cosmic_eric20140313.fcl -n 5
```

Then do something else.

callgrind (valgrind tool) in action (I)



callgrind (valgrind tool) in action (II)



CPU profiling summary

- tools provide complementary information
- tools information can be deceiving (or plainly wrong)
- once the bottleneck is detected, still work to be done!

Grampa's hints:

- beware of loops with a lot of iterations
  - put only what strictly needed, pull the rest outside
  - dont' write messages there (even `mf::LogDebug()` ones, since the message is still processed)

Example (I)

This is where `gperftools` is pointing to a nested `for` loop:
`larreco/RecoAlg/SpacePointAlg.cxx`, line 1267

```
for(std::map<unsigned int, art::Ptr<recob::Hit> >::const_iterator
      ihit2 = hitmap[cstat][tpc][plane2].lower_bound(wmin);
    ihit2 != hitmap[cstat][tpc][plane2].upper_bound(wmax); ++ihit2) {

  int wire2 = ihit2->first;
  const art::Ptr<recob::Hit>& phit2 = ihit2->second;

  // Get corrected time of second hit.

  double t2 = phit2->PeakTime() - detprop->GetXTicksOffset(plane2,tpc,csta
```

Example (II)

Pulling out stuff from the loop:

```
const double TicksOffset2 = detprop->GetXTicksOffset(plane2,tpc,cstat);

std::map<unsigned int, art::Ptr<recob::Hit> >::const_iterator
  ihit2 = hitmap[cstat][tpc][plane2].lower_bound(wmin),
  ihit2end = hitmap[cstat][tpc][plane2].upper_bound(wmax);
for(; ihit2 != ihit2end; ++ihit2) {

  int wire2 = ihit2->first;
  const art::Ptr<recob::Hit>& phit2 = ihit2->second;

  // Get corrected time of second hit.

  double t2 = phit2->PeakTime() - TicksOffset2;
```

might help (or not).
In this case, acting in this way in 4 different points of the code, `SpacePointFinder` became 20% faster and `Track3DKalmanHit` 15%, while the other modules fluctuated on $\pm 2\%$. On the event the gain was 10%.

Memory usage profiling

Goal:

Map the memory usage to the places where it is requested, and analyse its evolution in time.

Memory usage needs to be controlled in environments with limited resources (e.g., all the CPU farms).

- large pools of memory might be reduced by on-demand loading
- redesign of algorithms can reduce the need for caching (e.g., convert from one loop to load everything and one to analyse to one loop to load and analyse each event)
- constant increase of memory without its release can be identified and fixed
- sudden peaks of memory usage can be smoothed
- memory fragmentation can be mitigated

Memory profiling tools

`pmap` (Linux command): total memory map

`SimpleMemoryCheck` (`art` service): monitor per-event, module-level memory increase

`massif` (`valgrind` tool) records each allocation and deallocation

There are, as usual, more tools (for example, other `valgrind` tools, `memcheck` and `dhat`, and `gperftools` also has memory check capabilities),

Linux memory map (pmap)

Linux knows how much memory it is allowing the programs to run. This information can be accessed by:

from procfs `/proc/PID/maps` "file" gives a picture of the memory used by process number *PID*

`AddMemoryMap.py` (a script of mine) does the same (more control)

`pmap` has different options for the same thing

If you are running MacOS, there must be other ways (which I ignore).

Out of `AddMemoryMap.py -s mapsize /proc/PID/maps`:

```
    [...]
  676212 KiB    25012 KiB | /usr/lib64/libicudata.so.52.1
 1523792 KiB   847580 KiB | [heap]
 2794028 KiB  1270236 KiB |
/proc/PID/maps: 2861084672 bytes (2728.54 MiB) in 1099 pages and 250 group
```

with the first column the total so far, the dynamic memory shown in blue and the total memory usage in red.

art SimpleMemoryCheck in action

From `grep Memory LogFile.log` of a 10-event run log:

```
%MSG-w MemoryCheck:  Track3DKalmanHit:trackkalmanhit 14-Mar-2014 18:32:43
run: 1 subRun: 5 event: 42
MemoryCheck: module Track3DKalmanHit:trackkalmanhit VSIZE 1819.14 58.4375
%MSG-w MemoryCheck:  Track3DKalmanHit:trackkalmanhitcc 14-Mar-2014 18:34:1
run: 1 subRun: 5 event: 42
MemoryCheck: module Track3DKalmanHit:trackkalmanhitcc VSIZE 1831.12 11.984
%MSG-w MemoryCheck:  SpacePointFinder:spacepointfinder 14-Mar-2014 18:34:2
run: 1 subRun: 5 event: 42
MemoryCheck: module SpacePointFinder:spacepointfinder VSIZE 1831.96 0.8398
%MSG-w MemoryCheck:  Track3DKalmanSPS:trackkalsps 14-Mar-2014 18:34:34 CDT
run: 1 subRun: 5 event: 42
MemoryCheck: module Track3DKalmanSPS:trackkalsps VSIZE 1839.23 7.26172 RSS
[...]
%MSG-w MemoryCheck:  Calorimetry:trackkalspscalo 14-Mar-2014 18:53:50 CDT
run: 1 subRun: 5 event: 46
MemoryCheck: module Calorimetry:trackkalspscalo VSIZE 1888.11 5.03516 RSS
%MSG-w MemoryCheck:  RootOutput:out1 14-Mar-2014 18:53:50 CDT  run: 1 subR
MemoryCheck: module RootOutput:out1 VSIZE 1905.07 16.9688 RSS 1232.01 18.5
MemoryReport> Peak virtual size 1905.07 Mbytes
```

The peak memory can be wrong, but it's still informative.

massif (valgrind tool): example

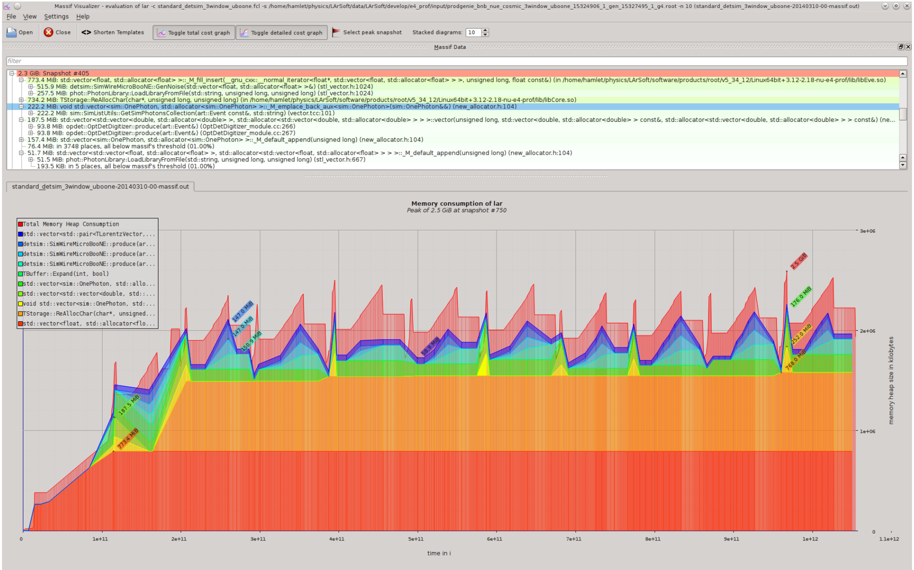
A detailed snapshot of a 2-event reco run writing a large event:

```
snapshot=951
#-----------
time=2755749742080
mem_heap_B=2222708828
mem_heap_extra_B=47481604
...
n10: 2222708828 (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
 n2: 1244677004 ...: TStorage::ReAllocChar(...) (in .../libCore.so)
  n4: 631258836 ...: TBuffer::Expand(int, bool) (in .../libCore.so)
   n1: 433599113 ...: TBasket::ReadBasketBuffers(long long, int, TFile*) (in .../libTree.so)
    ...
               n1: 166661418 ...: art::Ptr<raw::RawDigit>::getData_() const (.../Ptr.h:457)
                n1: 166661418 ...: trkf::SeedFinderAlgorithm::FindSeeds() (.../Ptr.h:344)
                 n1: 166661418 ...: trkf::SeedFinderAlgorithm::GetSeedsFromUnSortedHits(...) (.../S
                  n1: 166661418 ...: trkf::Track3DKalmanHit::produce(art::Event&) (.../Track3DKalma
                   ...
   n1: 166435561 ...: TBasket::WriteBuffer() (in libTree.so)
    ...
                n1: 166435561 ...: art::RootOutput::write(...) (.../RootOutput_module.cc:155)
```

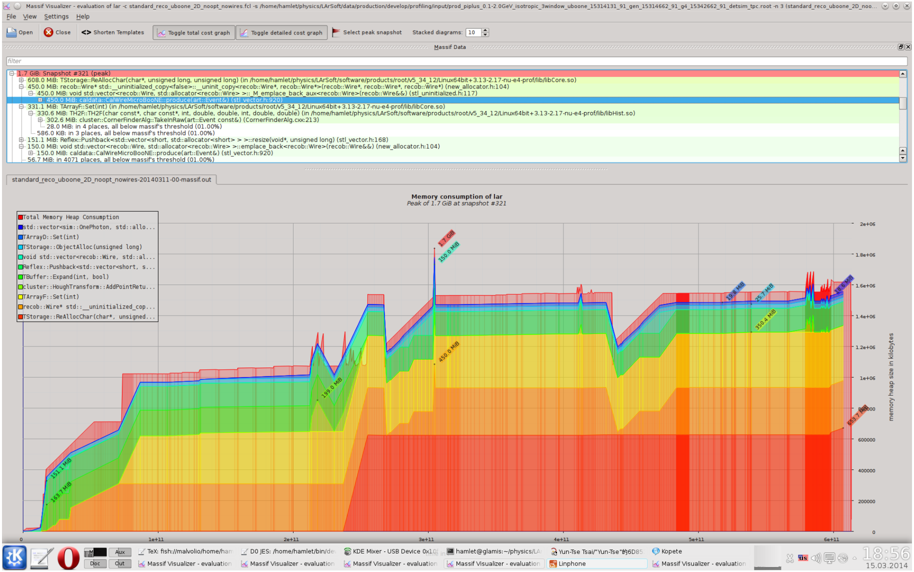
(ellipses are mine)

It says 2.2 GB of memory are allocated via malloc/new, more than half by ROOT tree (a good lot for writing, plus some for reading).

massif example: general shape (detsim)



`massif` example: memory leak + spike (reco 2D)



`massif` example: memory fragmentation? (reco 3D)