# Profiling memory and CPU usage

## with Google PerfTools and Valgrind

Brett Viren

May 30, 2025

This is meant to help the DNNROI inference optimization work.

# Overview

It is very difficult to understand memory and CPU usage by observing the the job as a **monolith**.

- Eg, by watching the program "`top`".
- One can **profile** the job to better understand which parts deserve "blame".

## How do you "profile" a job?

- Run the job with some **profiling tool**.
- The tool generates some output.
- Run a provided analysis tool to make a visualization.
- Examine the visualization to gain understanding.

## Google's PerfTools and Valgrind

These are both simple, commonly available and can be very useful.

# Run Google PerfTools

For shorthand:

```
$ export PROG=/path/to/your/program
```

## CPU

```
$ export CPUPROFILE=cpu.prof
$ LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libprofiler.so.0 $PROG [args]
$ ls -l cpu.prof*
```

## Memory

```
$ export HEAPPROFILE=mem.prof                          # required
$ LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libtcmalloc.so.4 $PROG
$ ls -l mem.prof*
```

# Visualize PerfTools results

Here we use the **cpu** file. The **mem** file works identically but is named like `mem.prof.heap.XXXX`.

### Basic graph generation

```
$ google-pprof --pdf $PROG $CPUPROFILE > cpu.pdf
```

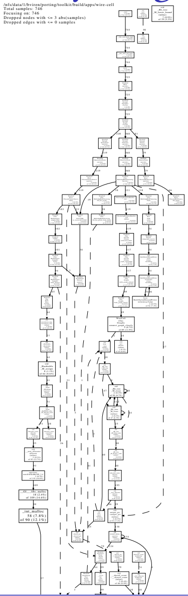
### Focus on a particular function:

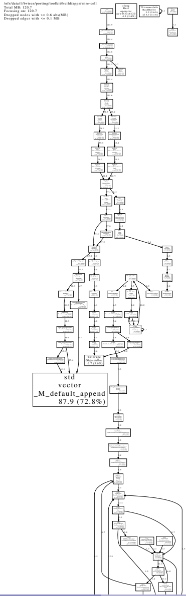
```
$ google-pprof --focus 'as_pctree' --pdf $PROG $CPUPROFILE > cpu.pdf
$ google-pprof --help
```

### More ways to process see:

```
$ google-pprof --help
```

# Example graphs - CPU and Memory usage


/nfs/data/1/bviren/porting/toolkit/build/apps/wire-cell
Total samples: 746
Focusing on: 746
Dropped nodes with <= 3 abs(samples)
Dropped edges with <= 0 samples
std _Rb_tree _M_lower_bound (inline) 7 (0.9%) of 39 (5.2%)
__GI___libc_malloc 18 (2.4%) of 109 (14.6%)
_int_malloc 58 (7.8%) of 90 (12.1%)


/nfs/data/1/bviren/porting/toolkit/build/apps/wire-cell
Total MB: 120.7
Focusing on: 120.7
Dropped nodes with <= 0.6 abs(MB)
Dropped edges with <= 0.1 MB
clang Decl operator new@37c6120 4.3 (3.6%)
TStreamerInfo ReadBuffer 3.2 (2.6%) of 3.7 (3.1%)
TStorage ObjectAlloc 6.7 (5.6%)
std vector _M_default_append 87.9 (72.8%)

- Nodes sized by how mcuh resource (cpu/mem) they use.
  - Numbers give used resource directly by the function or by functions the function calls.
- Edges show how much resource went for each (sub) function call.

Tips: Use a PDF viewer that allows easy zoom/pan and play with filtering and other `google-pprof` options.

# How Valgrind works

Valgrind works similarly to Google Perftools. It has a number of “tools”

callgrind makes snapshots of the call stack to make and analyze a call graph.

massif checks heap memory by overriding `malloc` etc and taking snapshots. Generates **profile file** for later visualization.

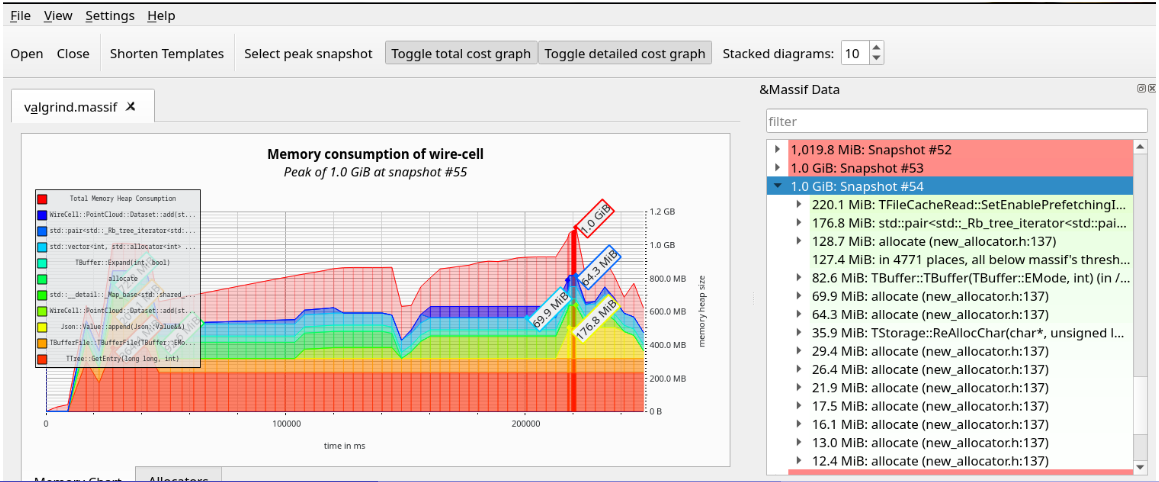
memcheck looks for memory leaks and other errors. Generates **summary report** and error messages.

General usage:

```
valgrind --tool=TOOLNAME [tool options] your_program [program args]
```

# Visualize massif results

```
$ valgrind --tool=massif --time-unit=ms --detailed-freq=1 --massif-out-file=v
  myprogram [my program args]
$ massif-visualizer valgrind.massif
```


File
View
Settings
Help
Open
Close
Shorten Templates
Select peak snapshot
Toggle total cost graph
Toggle detailed cost graph
Stacked diagrams: 10
valgrind.massif
&Massif Data
filter
Memory consumption of wire-cell
Peak of 1.0 GiB at snapshot #55
Total Memory Heap Consumption
WireCell::PointCloud::Dataset::add(st...
std::pair<std::_Rb_tree_iterator<std:...
std::vector<int, std::allocator<int> ...
TBuffer::Expand(int, bool)
allocate
std::__detail::_Map_base<std::shared_...
WireCell::PointCloud::Dataset::add(st...
Json::Value::append(Json::Value&&)
TBufferFile::TBufferFile(TBuffer::EMo...
TTree::GetEntry(long long, int)
1.0 GiB
64.3 MiB
69.9 MiB
176.8 MiB
1.2 GB
1.0 GB
800.0 MB
600.0 MB
400.0 MB
200.0 MB
0 B
memory heap size
0
100000
200000
time in ms
Memory Chart
Allocators

- 1,019.8 MiB: Snapshot #52
- 1.0 GiB: Snapshot #53
- 1.0 GiB: Snapshot #54
  - 220.1 MiB: TFileCacheRead::SetEnablePrefetchingI...
  - 176.8 MiB: std::pair<std::_Rb_tree_iterator<std::pai...
  - 128.7 MiB: allocate (new_allocator.h:137)
  - 127.4 MiB: in 4771 places, all below massif's thresh...
  - 82.6 MiB: TBuffer::TBuffer(TBuffer::EMode, int) (in /...
  - 69.9 MiB: allocate (new_allocator.h:137)
  - 64.3 MiB: allocate (new_allocator.h:137)
  - 35.9 MiB: TStorage::ReAllocChar(char*, unsigned l...
  - 29.4 MiB: allocate (new_allocator.h:137)
  - 26.4 MiB: allocate (new_allocator.h:137)
  - 21.9 MiB: allocate (new_allocator.h:137)
  - 17.5 MiB: allocate (new_allocator.h:137)
  - 16.1 MiB: allocate (new_allocator.h:137)
  - 13.0 MiB: allocate (new_allocator.h:137)
  - 12.4 MiB: allocate (new_allocator.h:137)

# Going further

GPU profiling has its own tools, usually specific to the hardware.

- NVIDIA: Nsight, nvprof, nvpp (nvidia-smi gives a “top” for the GPU)

Various profiling tools work at Linux kernel level.

- perf, eBPF, …

$\mathcal{FIN}$

# What is a call graph

Assume a program like:

```
void c() { c1_work(); c2_work() }
void b() { c(); b_work()  }
void a() { b(); a_work() }
void main() { a(); }
```

A snapshot looks at CPU/Memory usage in any instance and records the **call stack**.
Some example stack snapshots:

- `main() -> a() -> b() -> c() -> c1_work()`
- `main() -> a() -> b() -> b_work()`
- `main() -> a() -> a_work()`

Google Perftools simply book keeps how much CPU and Memory is being used on each snapshot.