CO339 – Tutorial on Profiling

Holger Pirk

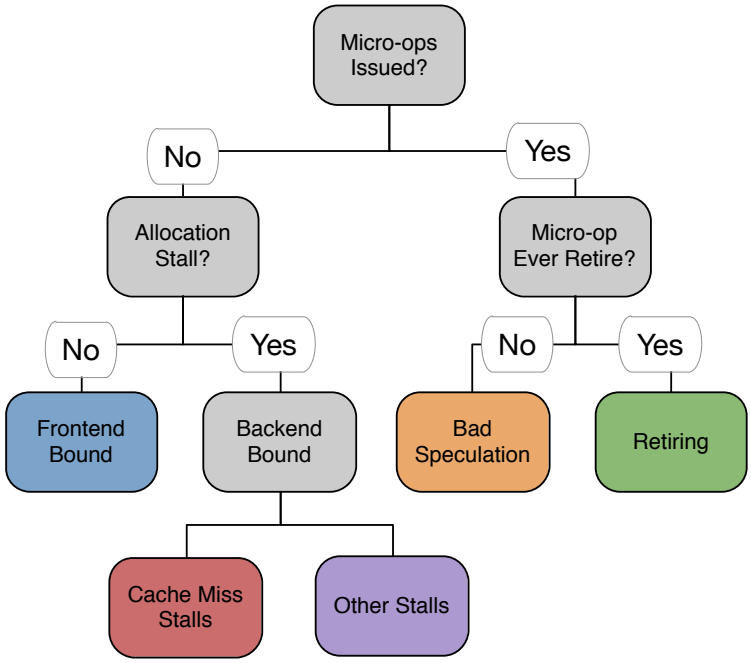
Welcome to the CO 339 tutorial on instrumentation and profiling. We are going to get familiar with Intel's VTune – a practical tool for profiling.

1 Preparation

1.1 Before class

- Download and install the free Intel VTune Profiler on your home computer or laptop. You can find it here (you have to install the oneAPI Base toolkit but we only care about the VTune Profiler component): `https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit/download.html`
  - Note that you do not have to register for the download (there is an inconspicuous link at the bottom of the "Sign In to Get Your Download" dialog)
- Make sure you can remote-login to a college machine of your choice (you might need to install an openvpn client): `https://www.doc.ic.ac.uk/csg/services/linux`
- Enable ssh login using public-keys: `https://www.linode.com/docs/security/authentication/use-public-key-authentication-with-ssh`
- Make sure you can edit files on the remote machines
  - I personally use emacs (either through a terminal or using tramp) but as I understand, both VSCode as well as CLion support remote editing through ssh (some people even use vi :-)

1.2 First things first: here is that tree again



- Do you think you can explain what it means?

1.3 Claim a machine

- Pick a 'vertex'-machine based on your breakout room number (so, if your room number is 17, use 'vertex17.doc.ic.ac.uk'). Single digit room numbers need a leading zero.

1.4 ssh into your machine

- It will be helpful if you can edit files directly on the remote machine

2 Profiling Demo

In this part, we are going to perform a quick bottleneck analysis for pe-grep. We are using the "hotspot" (sampling) profiler as well as the microarchitectural breakdown as described by the analysis tree (VTune actually has this built in now).

2.1 Watch me do it

```
cd $(mktemp -d)
curl --compressed https://www.gutenberg.org/cache/epub/2229/pg2229.txt | iconv -c -f UTF8 -t ASCII | tr -d '\r' > faust.txt
for i in {1..3000}; do cat faust.txt >> faust3000.txt; done


curl -OL https://co339.pages.doc.ic.ac.uk/Code/pegrep.cpp
clang++ -O3 -g pegrep.cpp
./a.out faust3000.txt

pwd # copy this path for vtune
```

Sit back and enjoy.

2.2 A problem

VTune has no understanding of the Google Benchmark Framework. It doesn't know that we don't want to profile the setup code. However, it has an API/library to control the tracing at runtime: ITTNotify.

2.3 Pinning it down using ITTNotify (on the server)

```cpp
#include "ittnotify.h"
int main(int argc, char* argv[]) {
  static auto domain = __itt_domain_create("somedomain");
  doSomeExpensivePreparation(); // replace with your code
  __itt_resume();
  for(size_t run = 0; run < runs; run++) {
    char taskname[128];
    snprintf(taskname, 128, "run number: %lu", run);
    auto task = __itt_string_handle_create(taskname);

    __itt_task_begin(domain, __itt_null, __itt_null, task);
    doSomeHeavyLifting(); // also replace with your code
    __itt_task_end(domain);
  }
  __itt_pause();

  return 0;
}
```

2.4 Compiling with ITTNotify (on the server)

```
curl -OL https://co339.pages.doc.ic.ac.uk/Code/pegrepitt.cpp
clang++ -I/vol/intel/parallel_studio_xe_2020/vtune_profiler_2020.3.0.612611/include -g -O3 pegrepitt.cpp
↪  /vol/intel/parallel_studio_xe_2020/vtune_profiler_2020.3.0.612611/lib64/libittnotify.a -ldl

# vtune

curl -OL https://co339.pages.doc.ic.ac.uk/Code/pegreplarge.cpp
clang++ -I/vol/intel/parallel_studio_xe_2020/vtune_profiler_2020.3.0.612611/include -g -O3 pegreplarge.cpp
↪  /vol/intel/parallel_studio_xe_2020/vtune_profiler_2020.3.0.612611/lib64/libittnotify.a -ldl

# more vtune
```

3 Now, it is your turn...

Now, you are going to perform a bottleneck analysis of your own. For that, we are profiling a data structure that might be familiar to some of you: an implementation of a B-Tree.

Keep in mind, that you do not have to understand the existing code. In fact, much performance analysis is performed exactly to avoid having to understand foreign, often large codebases. Let's get started...

3.1 Get the Btree sources on the server

Log in to your server and run

```
cd $(mktemp -d)
git clone https://gitlab.doc.ic.ac.uk/co339/BTrees.git # use your imperial credentials
cd BTrees/
mkdir build
cd build/
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
make -j8
./BTrees # play around with the parameters (nodesize, number of elements, number of runs) -- make in run in a second or so
```

3.2 Get the Btree sources on your client

- You only need those if you want to do analysis at a line-of-code granularity

3.3 Focus on the Driver (Main.cpp)

- Don't worry about the implementation in BTrees.cpp – it is a balanced tree index with configurable node fanout (maximum number of children per node)
- If you really need to know: `https://en.wikipedia.org/wiki/B-tree`

3.4 Spin up your VTunes (on your laptop)...

1. Objectives
   (a) Step 1
      - Identify the critical code sections of the program using "Hotspot Analysis"

   (b) Step 2 Perform a microarchitectural bottleneck analysis for the lookup (the "count" operation) phase. Focus on two questions:
      - Does the performance profile depend on the size of the tree (i.e., the dataset size)?
        - Hint: trees of about 128*1024 elements seem to fit in the cache
      - Does the bottleneck depend on the size of a node in the tree?
        - I implemented power-two node sizes from 4 through 128

   Hints:
      - Make sure you isolate the lookup part (using the ittnotify api)
      - In practice, it is hard to definitively identify a single microarchitectural bottleneck because of complex interactions of the different pipeline phases (an instruction that is stalled on execution port allocation could still be stalled on data access).
      - Discuss in your group and let us come up with an overall answer

4 Macro-Profiling

Let's focus on a real application now... Let's figure what SQLite's bottlenecks are during query execution. For that, we will use a simple graph path query.

4.1 Generate some random adjacency data

We'll use bash to generate some edges.

```
cd $(mktemp -d)
bash -c 'for i in {1..1678120}; do echo $[i*997%499],$[i*599%509]; done > /tmp/graph.csv'
```

4.2 Spin up sqlite (on the server)

```
sqlite3 -csv
```

4.3 Load the data into a table

```
create table graph (edgein int, edgeout int);

.import /tmp/graph.csv graph
```

4.4 Run a path query

The query simply counts the number of paths of length 2 between two nodes.

```
select count(*) from graph g1,graph g2, graph g3 where g1.edgein=4 and g1.edgeout = g2.edgein and g2.edgeout = g3.edgein
↪  and g3.edgeout = 417;
```

Now, use VTune's attach to process capability (behind the three dots in the "WHAT" section of the configuration) to attach to the process in Microarchitectural Exploration mode. Attach to `sqlite3` in paused mode, unpause, run the query (maybe a couple of times) and stop the tracing. To study one run of the query you can switch to the "Bottom Up" perspective, select a section of the timeline (the periods of high CPU usage are brown) and "Filter in by Selection". As VTune cannot identify the functions, you want to switch the "Grouping" to something that does not start with "Function /" – "Process / Thread / Module / Function / Call Stack" would be a good candidate.

- What do you observe?
- Explain & Discuss