Lab 10: Profiling

Comp Sci 1585
Data Structures Lab:
Tools for Computer Scientists

Outline

1 Introduction

2 time

3 callgrind

4 massif

5 Other options
gperftool
perf

Profiling

- Profiling ("program profiling", "software profiling") measures the empirical space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls.
- Most commonly, profiling information serves to aid program optimization.
- Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a profiler (or code profiler).
- Profilers may use a number of different techniques, such as event-based, statistical, instrumented, and simulation methods.

Profiling

Profiling measures the performance of a program and can be used to find CPU or memory bottlenecks.

- `$ time` A bash stopwatch
- `$ callgrind` Valgrind's CPU profiling tool
- `$ massif` Valgrind's memory profiling tool
- `$ Linux-perf` Linux profiling with performance counters
- `$ gperftool` Google performance tools
- `$ gprof` The GNU (CPU) Profiler

Outline

1 Introduction

2 time

3 callgrind

4 massif

5 Other options
gperftool
perf

Timing programs with `time`

- Just run `$ time ./your_program arg1 arg2 argn`
- Reading `time`'s output:
  - **Real**: The wall-clock or total time of execution
  - **User**: The time the program (and libraries) spent executing CPU instructions
  - **System**: The time the program spent waiting on system calls (usually I/O)

Timing programs with `time`

To improve the accuracy by taking the average across many runs:

```
#!/usr/bin/env bash
n=0
for run in {1..10}
do
        n=n+1
        temp_var=$(time ./your_progrom)
        # time_extract = write code to grab time
        time=time+time_extract
done
echo your_time is: $(( time / n ))
```

Note: this needs to be fleshed out.

Outline

1 Introduction

2 time

3 callgrind

4 massif

5 Other options
gperftool
perf

Profiling with `callgrind`

- As with Memcheck, compile with
  `$ g++ -g program.cpp -o program`
- Run `$ valgrind --tool=callgrind ./program`.
  It will create a file named `callgrind.out.NNNN`.
- `$ callgrind_annotate --auto=yes callgrind.out.NNNN`
  will print some statistics on your program.
  Redirect this into a file by appending `&>cg.txt`
- `$ kcachegrind callgrind.out.NNNN` reads profiling information and displays profiling statistics!
- You can also view the output file directly, although the results are not easy to read.

Understanding `callgrind` Output

- Callgrind counts instructions executed, not time spent.
- The annotated source shows the number of instruction executions a specific line caused.
- Function calls are annotated on the right with the number of times they are called.

Recursion and `callgrind`

- Recursion can confuse both `gprof` and `callgrind`.
- The `--separate-recs=N` option to Valgrind separates function calls up to N deep.
- The `--separate-callers=N` option to Valgrind separates functions depending on which function called them.
- In general, when you have recursion, the call graph and call counts may be wrong, but the instruction count will be correct.

Outline

1 Introduction

2 time

3 callgrind

4 massif

5 Other options
gperftool
perf

Profiling with `$ massif`

- Compile with `$ g++ -g program.cpp -o program`
- `$ valgrind --tool=massif --time-unit=B ./program` to run.
  It will create a file named `massif.out.NNNN`.
- To get information on stack memory usage as well, include `--stacks=yes` after `--time-unit=B`.
- `$ ms_print massif.out.NNNN` will print statistics for you.
- `$ massif-visualizer massif.out.NNNN` will show a much nicer interface
- To make every snapshot detailed, add: `--detailed-freq=1`

Understanding `massif` Output

- Snapshots: `massif` takes a snapshot of the heap on every allocation and deallocation.
  - Most snapshots are **plain**. They record only how much heap was allocated.
  - Every 10th snapshot is **detailed**. These record where memory was allocated in the program.
  - A detailed snapshot is also taken at peak memory usage.
  - By default, at most 100 snapshots are taken.
- The graph: Memory allocated vs. time. Time can be measured in milliseconds, instructions, or bytes allocated.
- Colons (:) indicate plain snapshots, 'at' signs (@) indicate detailed snapshots, and pounds (#) indicate the peak snapshot.
- The chart shows the snapshot number, time, total memory allocated, currently-allocated memory, and extra allocated memory.
- The chart also shows the allocation tree from each detailed snapshot.

Outline

1 Introduction

2 time

3 callgrind

4 massif

5 Other options
gperftool
perf

Outline

1 Introduction

2 time

3 callgrind

4 massif

5 Other options
gperftool
perf

gperftool

- Compile with: `$ g++ -g -lprofiler`
- Run your program:
  - Set the `CPUPROFILE` environment variable to the name of the file to store profile results in.
  - Then, run your program like normal.
  - For example,
    ```
    $ CPUPROFILE=gperftool.prof ./my-exe
    ```
- Use `$ pprof` to convert your output into cachegrind format:
  ```
  $ pprof --callgrind ./my-exe gperftool.prof > gperftool.out
  ```
- `$ kcachegrind gperftool.out` displays profiling statistics!

Outline

1 Introduction

2 time

3 callgrind

4 massif

5 Other options
gperftool
perf

perf

- `perf` began as a tool for using the performance counters subsystem in Linux, and has had various enhancements to add tracing capabilities.
- `$ perf stat -B ./myProg arg1 arg2`
- Tutorial:
  `https://perf.wiki.kernel.org/index.php/Tutorial`