Performance Analysis Tools

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah
Topics

- Tools for optimization
- What we can measure
- Measurement and analysis tools & techniques
Tools for Optimization
Tools for optimizing your code

- **Tools for creating fast code**
  - Optimizing compilers
  - Tuned software libraries

- **Tools for analyzing performance**
  - Timing tools
  - Profilers
  - Simulators / emulators
  - Assembler source code → read it!

- **Instrumentation**
  - Add instrumentation to your code
  - OS- or library-provided instrumentation tools
Optimizing compilers

- Modern compilers work hard to generate efficient code
- Many compilers let you choose an optimization level:
  - gcc -c -O1 myprog.c  \(\rightarrow\) O1 means basic, reasonable optimization
  - gcc -c -O2 myprog.c  \(\rightarrow\) O2 Try hard: good for production work
  - gcc -c -O3 myprog.c  \(\rightarrow\) O3 Go crazy!
- **Why not always use** O3?  
  - Takes longer to compile
  - Code can be hard to debug (some source lines disappear – can’t set breakpoint!)
  - Some years O3 is buggy 😞
- **Why not write assembler?**  
  - The compiler can do tricky things you likely would not
  - As CPUs change, compilers can change code generation strategies: your code still runs fast!
Using optimized libraries

- Highly optimized libraries are available for many purposes:
  - Sorting
  - Data query
  - Mathematical transformations (trig functions, Fourier transforms, etc.)
  - Etc., etc.

- Many of these are optimized far better than the code you are likely to write

- Where to get them:
  - Built into your OS or language runtime
  - Open source projects
  - Commercial offerings
What We Can Measure
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

```bash
$ time quick_sort < rand1000000
```

Built in shell `time` command

/`usr/bin/time` is a similar alternative
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

```
$ time quick_sort < rand1000000
1.116u 0.077s 0:01.25 94.4% 0+0k 0+0io 0pf+0w
```
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

```
$ time quick_sort < rand1000000
1.116u 0.077s 0:01.25 94.4% 0+0k 0+0io 0pf+0w
```
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

```bash
$ time quick_sort < rand1000000
1.116u 0.077s 0:01.25 94.4% 0+0k 0+0io 0pf+0w
```
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

```
$ time quick_sort < rand1000000
1.116u 0.077s 0.01.25 94.4% 0+0k 0+0io 0pf+0w
```
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

```
$ time quick_sort < rand1000000
1.116u 0.077s 0:01.25 94.4%     0+0k 0+0io 0pf+0w
```

This program took 1.25 seconds to complete.
The CPU was working on the program 94.4 % of that time.
1.116 seconds of CPU time was spent in the user’s own code.
0.077 were spent in the operating system doing work for the user (probably code to support file input/output)
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems
What are we measuring

- **Time**
  - Elapsed time $\rightarrow$ “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

Your program can also include timing checks *in the code* to measure specific sections. For more information, explore:

```
man 2 times
man 3 clock
man 2 getrusage
man 2 clock_gettime $\leftarrow$ used in COMP 40 CPUPTime_T
```
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

- **I/O (amount read/written and rate at which I/O is done)**

- **Paging rates (OS writes data to disk when memory is over-committed)**

- **Contention from other users**
OS Performance Displays
Measurement and Analysis
Tools & Techniques
Tools for understanding what your code is doing

- **Statistics from the OS:**
  - Overview of resources your program uses
  - Examples: `/usr/bin/time` command; `top`; etc. many others including info in `/proc`
  - Also look for tools that can report on I/O or network activity if pertinent
Tools for understanding what your code is doing

- **Statistics from the OS:**
  - Overview of resources your program uses
  - Examples: `/usr/bin/time` command; `top`; etc. many others
  - Also look for tools that can report on I/O or network activity if pertinent

- **Profilers:**
  - Typically bill CPU time (or other resources) to specific functions or lines of code
  - Examples: `gprof` (older, but simple & fast to use); `kcachegrind`
gprof

- Very traditional tool for finding bottlenecks
- Compiling for profiling:
  
  ```
  gcc -pg -o someprog someprog.g
  ```
- Invoke program normally (writes `gmon.out` performance file)
- To display performance summary:
  
  ```
  gprof someprog
  ```
- Moderately low overhead
- Level of detail barely adequate for modern performance analysis
kcachegrind

- Uses valgrind to instrument program:
  
  ```
  valgrind --tool=callgrind --dump-instr=yes someprog
  ```
  (creates `callgrind.out.xxx` file)

- Graphical interactive display of performance results:
  
  ```
  kcachegrind callgrind.out.xxx
  ```

- Very high overhead, but very useful detail down to machine instruction level
kcachegrind analysis of a quick sort
Tools for understanding what your code is doing

- **Statistics from the OS:**
  - Overview of resources your program uses
  - Examples: `/usr/bin/time` command; `top`; etc. many others
  - Also look for tools that can report on I/O or network activity if pertinent

- **Profilers:**
  - Typically bill CPU time (or other resources) to specific functions or lines of code
  - Examples: `gprof` (older, but simple & fast to use); `kcachegrind`

- **Instrument your code**
  - Add counters, etc. for code or I/O events that concern you
  - Use system calls à la `CPUTime_T` to do microsecond timing of important sections of code
  - Print output at end, or (more sophisticated) implement real time monitoring UI
  - *Make sure you understand whether your instrumentation is affecting performance of your code!*
Don’t measure tiny things

```c
/*
 * Measure time to add numbers
 */
double time_used;
int sum = 0;
for (i=0; i < 1000000; i++) {
    CPUTime_Start(timer);
    sum += arr[i]
    time_used += CPUTime_Stop(timer);
}
```

BAD:
(Time to invoke timer functions > time to add array element!)

```c
/*
 * Measure time to add numbers
 */
double time_used;
int sum = 0;
CPUTime_Start(timer);
for (i=0; i < 1000000; i++) {
    sum += arr[i]
}
time_used += CPUTime_Stop(timer);
```

MUCH BETTER:
(Only one call to timer…but we are also timing the loop code)
Tools for understanding what your code is doing

- **Statistics from the OS:**
  - Overview of resources your program uses
  - Examples: `/usr/bin/time` command; `top`; etc. many others
  - Also look for tools that can report on I/O or network activity if pertinent

- **Profilers:**
  - Typically bill CPU time (or other resources) to specific functions or lines of code
  - Examples: `gprof` (older, but simple & fast to use); `kcachegrind`

- **Instrument your code**
  - Add counters, etc. for code or I/O events that concern you
  - Use system calls to do microsecond timing of important sections of code
  - Print output at end, or (more sophisticated) implement real time monitoring UI
  - *Make sure you understand whether your instrumentation is affecting performance of your code!*

- **Read the generated assembler code**
Summary

- Understand the different reasons that programs run slowly
- Don’t guess, measure!
- Know how to use performance tools and understand what they tell you
- When necessary: read the assembler code

Next: we will explore techniques for rewriting your code to run faster.