Measurement and Tuning

Bradley C. Kuszmaul

MIT 6.172 — October 2, 2014
My Background

- MIT alum.
- **MIT Degrees = 2 × S.B. ’84 + S.M. ’86 + Ph.D. ’94.**
- Principal architect of the Connection Machine CM-5 supercomputer at Thinking Machines Corporation.
- Was an Assistant Professor at Yale.
- Worked at Akamai on the network usage database and network mapping.
- Cofounder of Tokutek, a high-performance database startup.
- Am a research faculty in the SuperTech group, working with Charles and Angelina.
- Reigning world champion for sorting a terabyte.
The Role of Measurement in Performance Engineering

- Implement a Correct Program
- Measure to Find the Problem
- Hack Performance Improvements

**Make it work.**
**Make it right.**
**Make it fast.**  
*Kent Beck*
Step 1: Start with a Correct Program

That sounds obvious.
But I mean a complete end-to-end solution.
With tests.

“Premature optimization is the root of all evil”
Donald E. Knuth
Part I: Make it Work

- Don’t start by optimizing the thing you “know” is the bottleneck.
- Instead, build a working solution.
- Do the bare minimum to make the test pass.
- Make the tests better.
- Sometimes you discover that you are working on the wrong problem, so you don’t have to optimize it.
- Often you discover that performance is dominated by something you didn’t expect.
Build Automated Tests

Building and testing your code should be as simple as

\$ make check

Speeding up code often reduces its readability. Optimizing broken code frustrates me.
Example Fib

fib.h:

```c
extern unsigned long fib (unsigned long n);
// Effect: Return \( F_n \), the \( n \)th Fibonacci number:
// \( F_0 = 0 \)
// \( F_1 = 1 \)
// \( F_N = F_{N-1} + F_{N-2} \) for \( N > 1 \).
```

Is that a good specification?

fib.c:

```c
#include "fib.h"

unsigned long fib (unsigned long n) {
    if (n<2) return n;
    else return fib(n-1)+fib(n-2);
}
```
Fib specification bugs

- What if the $n$th Fibonacci number doesn’t fit in an unsigned long?
- No performance requirements are specified.
Testing Fib

fib-test.c

#include <assert.h>
#include "fib.h"

int main (int argc,
       char *argv[] __attribute__((unused))) {
    assert(argc==1);
    assert(fib(1)==1);
    assert(fib(2)==1);
    assert(fib(3)==2);
    assert(fib(4)==3);
    assert(fib(5)==5);
}

Is this a good test?
A Makefile

Makefile:

```
fib-test: fib-test.o fib.o
check: fib-test
./fib-test
clean:
    rm -f *o fib-test
```

- First line: to build `fib-test`, first build `fib-test.o` and `fib.o`. Make knows the rules for compiling.
- Second line: defines a target `check`. First build `fib-test`. To build `check`, run `./fib-test`
Run it

$ make check
cc     -c  -o fib-test.o  fib-test.c
cc fib-test.o  fib.o  -o fib-test
./fib-test
make: *** [check] Error 5

What went wrong?
Forgot return value in `main()`

fib-test.c

```c
#include <assert.h>
#include "fib.h"
int main (int argc,
        char *argv[] __attribute__((unused))) {
    assert(argc==1);
    assert(fib(1)==1);
    assert(fib(2)==1);
    assert(fib(3)==2);
    assert(fib(4)==3);
    assert(fib(5)==5);
}
```

Got lucky: I happened to find the bug.
Can Tools Help Me Avoid This Error?

Makefile:

CFLAGS=-Wall -Werror
fib-test: fib-test.o fib.o
check: fib-test
    ./fib-test
clean:
    rm -f *.o fib-test

- First line turns on compiler warnings and makes them into errors.
- Turn on all the compiler warnings you can.
- Use serendipity. If you see an error, don’t let it slip away.
Run it again

$ make check
cc -Wall -Werror -c -o fib-test.o fib-test.c
fib-test.c: In function main:
fib-test.c:11:1: error: control reaches end of non-void function [-Werror=return-type]
ccl: all warnings being treated as errors
make: *** [fib-test.o] Error 1

I turned a run-time error into a compile-time error. That’s a good thing.
When I identified the bug, I wrote a test before I fixed the bug.

Q: What are the consequences of fixing the bug first?

A: The bug could come back and haunt me.
Fix return value in `main()`

fib-test.c

```c
#include <assert.h>
#include "fib.h"

int main (int argc,
          char *argv[] __attribute__((unused))) {
    assert(argc==1);
    assert(fib(1)==1);
    assert(fib(2)==1);
    assert(fib(3)==2);
    assert(fib(4)==3);
    assert(fib(5)==5);
    return 0;
}
```
Run it again

$ make check
c -c -o fib-test.o fib-test.c
cc fib-test.o fib.o -o fib-test
./fib-test

All this and we haven’t even started performing tuning! It’s worth it.
Don’t waste time optimizing broken code. Make it easy to test your code so that you can quickly check to see if your optimizations are broken.
Step 2: Measure the Problem

Roughly, 80% of the effects come from 20% of the causes. — The Pareto Principle.

- 80% of the land in Italy was owned by 20% of the population (Pareto).
- Worldwide 82.7% of the income goes to the top 20% of the population. (Human Development Report)
- 80% of a business’s profits come from 20% of the customers.
- Fixing 20% of the bugs solves 80% of the crashes (Microsoft)
- 80% of the runtime is spent in 20% of the code.
- 10% of the pages of memory account for 90% of the DRAM errors.
Don’t Optimize Infrequently Run Code

- If some code turns out to run only 1% of the time, how much speedup could you possibly get by speeding it up?
- Gene M. Amdahl argued that only a constant fraction of the work can be parallelized limiting the value of parallel computing. [“Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”, 1967.]
- Do you believe Amdahl’s argument?
Use the Tools

- Turn on every compiler warning you can.
- For example, Valgrind is a key tool for getting your program to work. Valgrind will complain about memory that you read without initializing, or accessing off the end of an array.
- Your programs should run with no memory warnings under Valgrind.
- Some memory allocation libraries have “debugging modes” that produce errors if you mess up.
- The Valgrind DRD tool will warn if you misuse the pthread API.
- Other tools?
A Recent Tokutek Valgrind Bug

```c
struct fileid {
    dev_t device;    // 8 bytes on linux, 4 bytes on OSX.
    uint64_t inode;  // 8 bytes.
};

bool compare_fileid (const struct fileid *a,
                     const struct fileid *b) {
    return memcmp(a, b);
}
```

On OSX, `memcmp()` looks at the 4 uninitialized bytes between `device` and `inode`. 
Build Automated Timing

- You must also make it easy to run timing tests.
- If you make a change that’s only worth 1% it can be tough to measure the improvement.
- Improving 0.1% is even harder to measure.
- But several small changes add up to a measurable improvement.
- Sometimes, performance is tough to quantify: Don Dailey, who worked with us on the world-class CilkChess Program, said that to tell whether a particular change is good requires playing hundreds of games. Must automate.
Use Version Control

- Sending yourself an email containing your source code?
- Invest the effort in learning a version control system such as \texttt{svn} or \texttt{git}.
- Put your code, documentation, and everything else under version control.
- We’ve seen people make changes, and then realize they had slowed things down, and so they tried to undo the changes, but the code was still slower. They couldn’t go back.
- Version control lets you make improvements with impunity. You can always go back.
Utilize Version Control

Don’t just use version control. *Utilize* it (that is, make efficient use of it).

- Take the time to write good commit messages, so you can figure out what a particular change does.
- Use an integrated bug tracking system.
- Keep a “laboratory notebook”. I like to collect the entire wallpaper from a performance measurement section and add it to the repository. Then I can see exactly what I did, and exactly what the computer printed.
Ways to Measure a Program

- External measurements — *time*.
- Instrument the program — Change the program to measure it. E.g., call `gettimeofday()`.
- Interrupt the program — Stop the program periodically, and look at its state. E.g., use `gdb`.
- Hardware and Operating Systems Support — run the program with counters maintained by the hardware and operating system. E.g., use `perf`. 
The `time` command can measure elapsed time, user time, and system time for an entire program. (Who knows what that means?)

Sometimes you want to measure just part of a program, however. For example, you might not want to measure the initialization of your test structures.
The `perf stat` command can measure other hardware events, such as the number of page faults, cache misses or mispredicted branches.

Sometimes you want to measure just part of a program, however. For example, you might not want to measure the initialization of your test structures.
gettimeofday()

To measure just part of your program:

```c
#include <sys/time.h>
...
// timeval has has two fields: tv_sec and tv_usec,
// which are seconds and microseconds, respectively,
// since the Epoch (1/1/1970).
struct timeval start, end;
gettimeofday(&start, NULL);
function_to_measure();
gettimeofday(&end, NULL);
double tdiff = (end.tv_sec-start.tv_sec)
             + 1e-6*(end.tv_usec-start.tv_usec);
```

Note: To use `gettimeofday`, you modify your program to measure performance.
\texttt{clock\_gettime(CLOCK\_MONOTONIC,...)}

- `gettimeofday()` considered harmful: the measured time does not always run at the same speed. Sometimes time even goes backwards.
- **Use `clock\_gettime(CLOCK\_MONOTONIC,...)`**.

```
#include<time.h>

\texttt{struct timespec start, end;}
\texttt{clock\_gettime(CLOCK\_MONOTONIC, \&start);}
\texttt{function\_to\_measure();}
\texttt{clock\_gettime(CLOCK\_MONOTONIC, \&end);}
\texttt{// Note: nanoseconds, not microseconds.}
\texttt{double tdiff = (end.tv\_sec-start.tv\_sec)}
\texttt{+ 1e-9*(end.tv\_nsec-start.tv\_nsec);}
```
Clock Overheads

- On my laptop, `clock_gettime(CLOCK_MONOTONIC)` takes 83ns.
- That’s two orders of magnitude faster than a system call.
- `strace` (which lists all the system calls a program makes) shows that no system calls are being made. How do they do it?
- Warning: sometimes these calls run very slowly.
x86 processors provide a hardware counter called the time stamp counter (TSC). Read TSC as:

```c
static __inline__ unsigned long long rdtsc(void)
{
    unsigned hi, lo;
    __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
    return ( ((unsigned long long)lo)
             | (((unsigned long long)hi)<<32));
}
```

The time returned is “clock cycles since boot”.
rdtsc() problems

- The counter may give different answers on different cores on the same machine. (Recent Intel hardware seems better.)
- The counter sometimes runs backward. (Some drivers hack the counter to hide their presence.)
- The counter may not progress at a constant speed. (Recent chips are spec’d to run at a constant speed, even if the processor clock speed varies.)
- Converting clock cycles to seconds can be ... tricky.
How can `clock_gettime()` be so fast?

- The C library and the Linux Kernel work together.
  - **Kernel**: maintains, in user-accessible memory, an offset \( o \) and a scaling factor \( f \).
  - **Library**: Computes \( t = o + \text{rdtsc}() \times f \).
- Per-process time (CLOCK_PROCESS_CPUTIME_ID) can be computed in userspace if the kernel reduces the offset to account for times that a process is descheduled.
- Library reads \( o \) once before the `rdtsc`, and once after, to see if the program was descheduled in between (which would invalidate \( o \)).
Just use `clock_gettime()`

- Many kernel developers say “Don’t use `rdtsc`, you’ll be sorry!”
- They say use `clock_gettime()`.
- `clock_gettime()` runs almost as fast (83ns) as `rdtsc()` (32ns). (I’d be interested in hearing about other machines.) And why is `rdtsc()` so slow?
- You will be less sorry.
libpfm4 virtualizes all the hardware counters

- Modern kernels make it possible for libraries such as libpfm4 to measure all the provided hardware event counters on a per-process basis.
- `perf stat` employs libpfm4.
- There are many esoteric hardware counters. Good luck figuring out what they all measure.
- Watch out: You probably cannot measure more than 4 or 5 counters at a time without paying a penalty in performance or accuracy.
Instrumenting with a compiler

The compiler can instrument your code, e.g., for `gprof`:

- Add `-pg` flag to your compiles and links.
- Run your program.
- Run `gprof` to get an analysis of where time was spent.

Gprof instrumentation causes your program to be interrupted to record the program counter 100 times per second.

Not accurate if you don’t get enough samples.
Interrupting

- Idea: Run your program under **gdb**, and type `control-C` periodically. Look at the stack. Pretty soon you can see where your program spends time.
- Who needs compiler instrumentation?
- Some call this the “poor man’s profiler”.
pmprof automates poor-man profiling

- Idea: Run your program, interrupt it frequently, collect the top few line numbers in the stack, and count the number of times each stack fragment appears, displaying the most frequent stack fragments first.

- Here is pmprof:

```awk
BEGIN { s = ""; }
/^Thread/ { if (s !~ "") print s; s = ""; }
/^\#/ { if ($3 == "in") { v = $4; } else { v = $2 } if (s !~ "") { s = s"," v} else { s = v } }
END { print s } |
```

```
sort | uniq -c | sort -r -n -k 1,1
```

- Facebook uses this to understand their multi-petabyte MySQL database system.
perf record

The Perf toolset also provides **perf record/** perf report**, which is like a mix of **gprof** and **perf**.

- You run with **perf record**, which interrupts your program and uses **libpfm4** to measure hardware events.
- Then you run with **perf report**, which indicates, for each line of code or each instruction, where the events occurred.
Simulation for Measurement

If the hardware and operating system don’t measure what you want, you can use simulation. For example, Cachegrind (a Valgrind tool) can simulate cache behavior to help you understand if your program makes good use of cache.

You can write your own tools using the Intel Pintool pintool.org.
Do you believe your data?

How do you know your measurements are accurate?
Target sheets for Two Sharpshooters

Which would you rather recruit?

(a)

(b)
Taguchi Process Improvement

Genichi Taguchi pioneered this process-improvement strategy:

1. Minimize variability.
2. Adjust output to hit target.

Applied to quality control, but also can apply to other fields, such as performance measurement.
Reducing System Variance

- Quiesce the system.
- Disable background jobs.
- Kick off other users. (Use a job queue system.)
- Put the machine behind a firewall, or disable the network, so network packets don’t interrupt.
- Don’t wiggle the mouse.
Reducing Hardware- and OS-Induced Variance

- Disable hyperthreading.
- Turn off variable-speed clocks. (E.g., turn off turbo-boost.)
- Use `taskset` to force your job to run on specific cores. If you have a 2-thread job and a 16-core 2-socket machine, you may get different answers if the two threads run on the different sockets, on the same socket, or on the same core.
Reducing Software-Induced Variance

- Isolate the part of the code you want to measure. Better to measure a small system than a big one.
- Align code and data on cache boundaries.
- Can you get rid of variance due to calls to \texttt{random()}? Save the random seed and reuse it.
Measurement Advice

- Make the long run enough. (Short runs are difficult to measure.)
- Watch out for the measurement perturbing the results.
- Directly measure what you think is important. Don’t try to estimate the number of cache misses by measuring time. Instead, measure the cache misses.
- If you think that there is an occasional event that perturbs your measurements, take the minimum of several measurements.
Measure Again

When your code gets faster, previously insignificant things start making a difference.
Measure Quality

Measurement isn’t always just about performance.

- Code quality can be measured.
- Measure the rate at which bugs are appearing or being fixed.
- Measure code coverage. The `gcov` tool can tell you which lines of code have not been executed, and which branches have not been taken.
- Akamai had a system called `diffcov` that combined the results of `svn diff` with the `gcov` output. They had a policy that if you changed code, you must demonstrate that the tests exercise the changed code.
Noise in Coverage Measurements

Redefine assert() to get rid of never-taken branches. (Don’t disable assert().)

// Don’t do this:
define old_assert(expr) if (!(expr)) __assert_failure(#expr, __FILE__, __LINE__)

// Do this
define assert(expr) __check_assertion(expr, #expr, __FILE__, __LINE__)

// Define this in a separate file.
#define __check_assertion(int expr, char *expr_string, char *file, char *line) {
    if (!expr) {
        fprintf(stderr, "%s:%d Assertion failed: %s\n", file, line, expr_string);
        abort();
    }
}

Modern compilers perform link-time-optimization.⇒ macros should be simplified.
Need a Performance Model

To tune performance, you need a performance model. What should you measure? How should you interpret it?

- Lines of code. (Some lines of code are more expensive than others.)
- C++ language primitives. (Some primitives are more expensive. Vectorizable instructions can make things cheaper.)
- Machine instructions: Some instructions are more expensive. (Square root compared to increment.)
- Machine instructions weighted by their cost. (Some instructions vary in cost: For example a load with a cache hit vs. a cache miss.)
- Cache Misses. TLB misses.
After measuring things, you can sometimes infer values with curve fitting.

Suppose I have this data:

```
#time   instructions  cachemisses   Program
34864   170889186565542  36615004052  python
2618.6  7509707536406    39322034007  java
1480    2274589361551    68047140354  C gcc -O0
430.55  278479001783     34049504541  C gcc -O3
```

I want to infer how long it takes to run an instruction, and how long to take a cache miss.
Fitting Idea

I’ll guess that I can model the run time $T$ as

$$T = a \cdot I + b \cdot C,$$

where

- $I$ is the number of instructions, and
- $C$ is the number of cache misses.
Using Gnuplot

$ gnuplot
> f(x,y) = a*x + b*y
> fit f(x,y) "data" using 2:3:1:(1) via a,b
...

Final set of parameters                             Asymptotic Standard Error
=============================================    ==========================

a = 2.00447e-10 +/- 2.312e-12 (1.153%)
b = 1.71202e-08 +/- 4.245e-09 (24.8%)

correlation matrix of the fit parameters:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>-0.422</td>
<td>1.000</td>
</tr>
</tbody>
</table>
$ gnuplot
...
Final set of parameters

<table>
<thead>
<tr>
<th></th>
<th>Asymptotic Standard Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2.00447e-10 +/- 2.312e-12 (1.153%)</td>
</tr>
<tr>
<td>b</td>
<td>1.71202e-08 +/- 4.245e-09 (24.8%)</td>
</tr>
</tbody>
</table>
...

An instruction costs $0.2 \pm 0.002\text{ns}$.
A cache miss costs $17 \pm 4\text{ns}$. (Lots of uncertainty there.)
Watch out: The asymptotic standard error can be misleading. RTFM.
Correlation Matrix

$ gnuplot
...
correlation matrix of the fit parameters:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>-0.422</td>
<td>1.000</td>
</tr>
</tbody>
</table>

... 

This matrix provides an indication of how hard it is for one parameter to compensate for another. The diagonals are all 1. If all the parameters were independent, the others would be 0. Here, \( a \) and \( b \) have negative correlation. Probably meaningless since we have too little data.
Overfitting

If I add more basis functions to the curve fit, I can make the curve fit better. How do I know if I’ve got too much?

- Removing basis functions doesn’t hurt the quality of the model.
- Ask: Is the model predictive?
Is the Model Predictive?

One way to tell is to train on random data.

- Pick half the data at random.
- Use that data to find the coefficients for the curve fit.
- Using those coefficients, find out how good the model predicts the other half of the data.

Don’t train on all the data, and then report how good the model is at predicting the same data! (I’ve seen this at conferences.)
Triangulate

- Measure your code several ways.
- Check that the different ways of measuring tell a story consistent with your model.
- Like in a spreadsheet in the days of pencil-and-paper: make sure that the sum of the rows adds up to the same answer as the sum of the columns.
To Tune: Reduce Uncertainty

- To tune your programs, you must find ways to reduce uncertainty
  - Remove bugs.
  - Remove noise, other users, OS artifacts.
  - Measure accurately.
  - Model the performance.
- Good tuning requires good measurements.
- Good measurements require creativity and work.