Making Programs Faster

Benchmarking, Performance Tuning, and Caching

Mark Jason Dominus

Plover Systems Co.

    mjd-tpc-perf+@plover.com

v1.2 (May, 2003)

Making Programs Faster

- What we'll do:
    - Basic concepts, example optimizations
    - Using profiling tools
        - Mail folder analyzer
        - `perldoc`
    - Blunders
- Along the way:
    - Building custom profiling tools
    - What not to worry about
    - More blunders

Performance Tuning is Hard

- You want your program to be faster
  - So you guess what it might be spending a lot of time on
  - Then you guess that a different design will spend less time
  - Then you implement your guess
- Then you find out that you were wrong
  - There are no experts here
  - Everyone guesses wrong
- Guessing doesn’t work

Performance Tuning is Hard

- With some things, a seat-of-the-pants approach works fine
  - Not performance tuning
- You must be scientific and methodical
- It’s easy to mess up
- This class is about *tools* and *measurement*

Schwartzian Transform

- Sort list of items by some *non-apparent* feature
- Example: Sort filenames by last-modified date
- The obvious method:

```
sort { -M $b <=> -M $a }
     (readdir D);
```

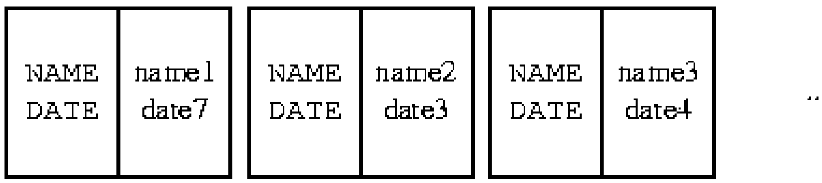
- It calls `-M` over and over on the same files
- Idea: Maybe we can speed this up as follows:

  1. Construct data structure with both names and dates
  2. Sort by date
  3. Throw away dates

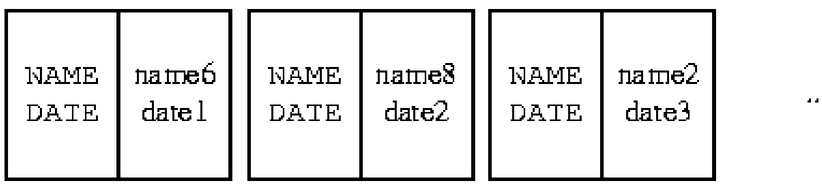
Schwartzian Transform

```
@names = readdir D;
```

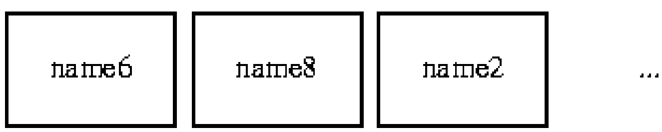

```
@names_and_dates =
  map { { NAME => $_, DATE => -M $_ } }
  @names;
```




```
@sorted_names_and_dates =
  sort { $b->{DATE} <=> $a->{DATE} }
  @names_and_dates;
```




```
@sorted_names =
  map { $_->{NAME} }
  @sorted_names_and_dates;
```



Schwartzian Transform

```
@sorted_names =
  map { $_->[0] }
  sort { $b->[1] <=> $a->[1] }
  map { [ $_, -M $_ ] }
  readdir D;
```

- This is more complicated and more work than the original code:

```
sort { -M $b <=> -M $a } readdir D;
```

- Is it really faster?
- To find out, we run both versions on the same data
  - We measure the time taken by each one
- This is called a *benchmark*

Schwartzian Transform

- On a sample of 11,632 files:

```
                User   Sys  Total
      Direct    0.80  2.55   3.35
 Schwartzian    1.14  0.39   1.53
```

- This says that the Schwartzian version was indeed about 54% faster for this example

Time

- The computer has several kinds of time
  - *Wallclock* time is actual elapsed time
  - On a timesharing system, this is rarely the amount of time the process actually spent working
  - It shares the processor with the OS and with other processes
- Of the wallclock time, some was spent executing instructions in the process's program
  - For example, copying data around or doing tests
  - This is the `user time`
- Some time was spent by the OS executing OS instructions at the program's request
  - For example, fetching mtimes, performing I/O, and allocating memory
  - This is the `system time`
- `user time` + `system time` = CPU time <= wallclock time

`user time` + `system time` = CPU time

Before

```
sort { -M $b <=> -M $a } readdir D;
```

After

```
@sorted_names =
  map { $_->[0] }
  sort { $b->[1] <=> $a->[1] }
  map { [ $_, -M $_ ] }
  readdir D;
```

- `-M` and `readdir` consume mostly system time
- Everything else is pure user time
- The goal of the Schwartzian Transform is to reduce the number of `-M`'s
  - But optimization is always a tradeoff
  - The cost is a lot more user-mode processing
  - We see this in the timing outputs

```
                 User   Sys  Total
      Direct     0.80  2.55   3.35
 Schwartzian     1.14  0.39   1.53
```

- The Schwartzian transform does 43% more processing
  - But it wins by asking the kernel for 84% less service

"Optimizations"

- The world is full of dumbassed 'optimizations' and 'benchmarks'
- We'll see several today
- Here's one I found while researching the Schwartzian Transform
- The goal here is to do a case-insensitive sort

```
sort { lc $a cmp lc $b } @stuff;
```

- Here's what was suggested:

```
Date: Sat, 15 Mar 1997 00:55:47 GMT
Subject: Re: Sorting help
Message-Id: <3329eefd.140372364@news.oz.net>

# The *drum roll* Schwartzian Transform!
@sorted = map  {$_->[0]}
          sort {$a->[1] cmp $b->[1]}
          map {[$_, lc $_]}
          @stuff;
```

- **Boldface** code is operations that were added

"Optimizations"

```
@sorted = map  {$_->[0]}
          sort {$a->[1] cmp $b->[1]}
          map  {[$_, lc $_]}
          @stuff;
```

- Here are the benchmark results on a list of 11,632 strings:

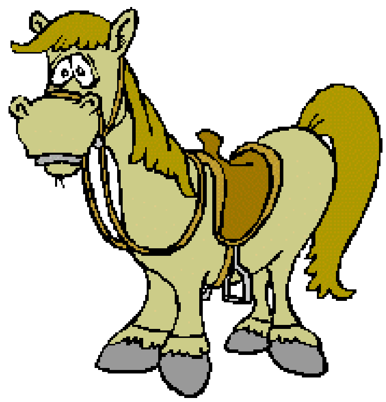
```
               User   Sys  Total
     Direct    0.23  0.00   0.23
 Schwartzian   0.85  0.08   0.93
```

"But I want a pony!"

```
                User   Sys  Total
      Direct    0.23  0.00   0.23
 Schwartzian    0.85  0.08   0.93
```

Performance tuning is *always a tradeoff*

- Never say "I'll use the Schwartzian Transform because it's faster"
    - That's an immature view of value
- That's what little kids are thinking when they say



Dad, can I have a pony?

- The poor little kid sees the benefit, but not the cost
- Always remember to ask

What am I spending and what am I getting in return?

- Unfortunately, the cost-benefit ratio for the pony is prohibitively large

"Optimizations"

```
sort { lc $a cmp lc $b } @stuff;

@sorted = map  {$_->[0]}
          sort {$a->[1] cmp $b->[1]}
          map  {[$_, lc $_]}
          @stuff;
```

- The 'benefit' here was to reduce the number of `lc` operations
  - The cost was to introduce array reference lookup operations in their place
  - And two extra scans over the list
  - And some memory allocation
  - But he got his pony!
- More ponies later

Wallclock Time

- Wallclock time is the most natural way to measure performance
- Because you want the program to finish sooner rather than later
- But measuring wallclock time directly is very tricky
    - Operating systems like Unix and Windows do pre-emptive multitasking
    - At any moment the OS might put any process to sleep for a long time
    - Processes go to sleep when the OS wants to do something else
    - Sleeping processes consume wallclock time but not CPU time

Wallclock Time

- If a program needs to do a certain amount of computation, that consumes a certain amount of CPU time
  - The amount of CPU time will probably not vary too much for a particular task
- However, wallclock measurements can vary a lot from one run to another
  - It all depends on what else is going on at the same time
  - The amount of wallclock time might vary enormously
  - Variations might be unrelated to the program you are examining
- For this reason we tend to concentrate on measuring CPU, which is easier

Wallclock Time

- Unfortunately, measuring CPU isn’t always what you want
- Consider a program with high wallclock time but low CPU time
    - This program is spending a lot of time waiting around
    - That may be unavoidable
    - Reducing the CPU usage of this program may not reduce its wallclock usage proportionally
    - It may be computing faster but spending the same amount of time waiting around

Wallclock Time

```
# usage: webgrep PATTERN urls...
use LWP::Simple 'get';
my $pat = shift;
my @contexts;
for my $url (@ARGV) {
  my $doc = get($url);
  unless (defined $doc) {
    warn "Couldn't fetch $doc; skipping\n";
    next;
  }

  while ($doc =~ m/$pat/oig) {
    push @contexts, substr($doc, pos($doc) - 30, 60);
  }
}
print join("\n--------\n", @contexts), "\n";
```

- This program's wallclock time is dominated by the call to `get`
  - `get` spends most of its time waiting for messages to travel across the network
  - We say that the program is *I/O bound*

I/O Bound Programs

- To speed up `webgrep`, we would need to address the network latency time
- It is unlikely that altering the search itself will produce much of an effect
- The benchmarks bear this out:

```
% ./webgrep perl http://www.perl.com/

real    0m3.456s
user    0m0.720s
sys     0m0.070s
```

- CPU time accounted for only about 23% in this simple case

```
% ./webgrep perl http://www.perl.com/ http://www.perl.com/ \
        http://www.perl.com/ http://www.perl.com/ \
        http://www.perl.com/ http://www.perl.com/

real    0m15.599s
user    0m0.840s
sys     0m0.110s
```

- 6% in this case
- Trying to speed up an I/O bound program by reducing the amount of computation won't work
- Alternative:
    - Parallelize I/O (asynchronous I/O; move it to subprocesses, etc.)

I/O Bound Programs

- CGI application performance is another great example of this
- When the user submits a form, the following happens:

    1. The browser sets up a TCP connection to the server
    2. It sends the form contents
    3. The server starts a new CGI process
    4. The process loads the CGI program and compiles it
    5. **The CGI program runs**
    6. The server gathers the CGI output and constructs a response
    7. It sends the response to the browser
    8. The connection is torn down
    9. The browser renders and displays the results

- All this typically takes a couple of seconds
- Speeding up the CGI program itself only speeds up step 5
- This probably has a minimal effect on the user's experience

CPU Bound Programs

- In contrast, consider this program:

```
for my $i (1 .. 100000) {
  my $n = $i / 100;
  my $s = square_root($n);
}

sub square_root {
  my $tolerance = 0.000001;
  my $g = my $n = shift;
  while (abs($g * $g - $n) >= $tolerance) {
    $g = ($n/$g + $g)/2;
  }
  $g;
}

real    0m10.211s
user    0m9.570s
sys     0m0.010s
```

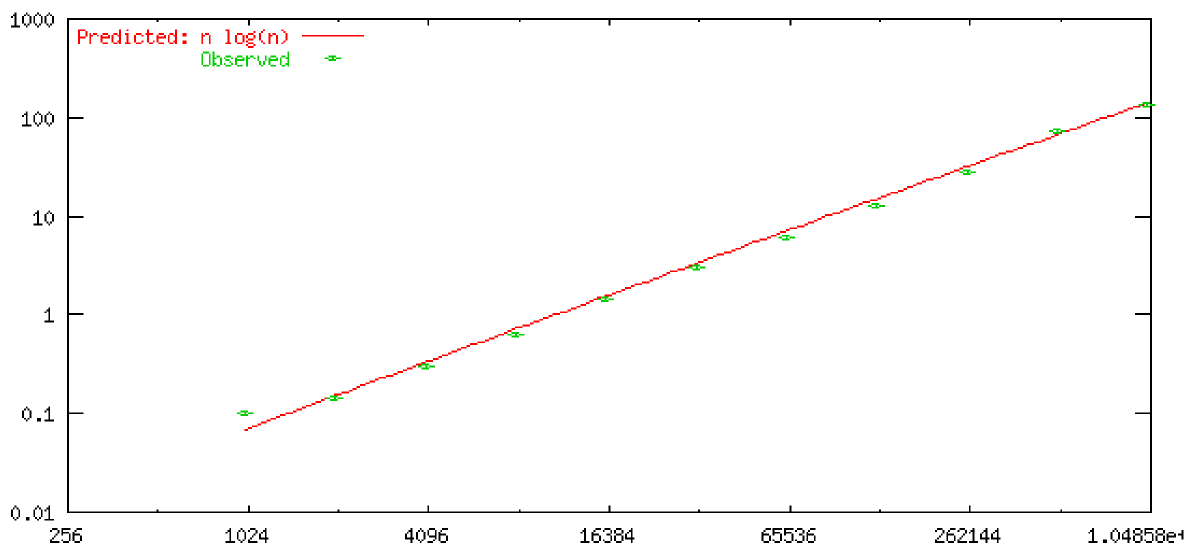
- This program spent 94% of its life using the CPU
  - Reducing the amount of computation by even 10% is likely to have a significant effect on the wallclock time
- We say such a program is *CPU bound*

Memory Bound Programs

- Some programs do relatively little computation or I/O but are slow anyway
- Consider this simple program:

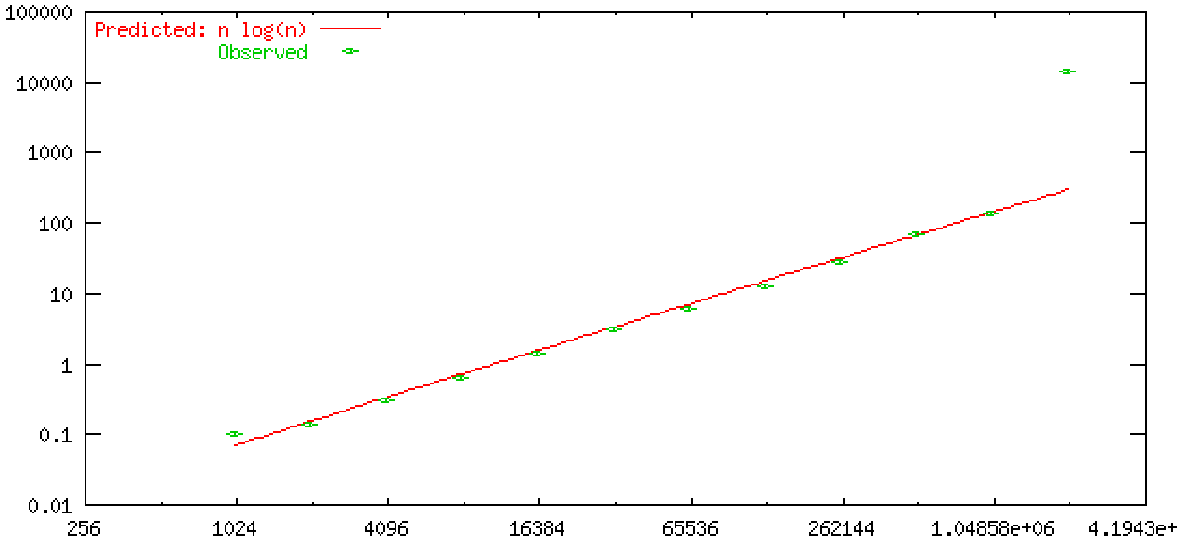
```
print sort <>;
```

- Theoretically, the `sort` runs in $O(n \log n)$ time, on average
    - That means that if the input size doubles, the run time should be a little more than twice as long



- From this we might extrapolate that 2048000 items will take about 283 seconds
- Actually it took 14,601 seconds

Memory Bound Programs



- What happened here?
- 1024000 items fit into real memory; 2048000 didn't
  - The OS had to start swapping pages to disk
  - Program run time was dominated by the swapping time
- For large input lists, this program is *memory bound*
  - Its slowness is caused not by excessive computation but by excessive memory usage
  - Performance will be most improved by reducing memory usage

Simple Measurement Tools

- Most Unix systems come with a command called `time`
- For quick estimates of entire programs, the `time` command is handy

```
% time ls
#BIGMESS#       PENN    YAPC_16.jpg     gym     photos
...
real    0m0.858s
user    0m0.130s
sys     0m0.230s
```

- Often this is built into the shell; the `time` program is different:

```
% /usr/bin/time ls
#BIGMESS#       PENN    YAPC_16.jpg     gym     photos
...
0.13user 0.23system 0:00.86elapsed 41%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (130major+24minor)pagefaults 0swaps
```

- `41%CPU` here is the *CPU utilization*
  - It's just CPU time divided by wallclock time
- Most of the other information is provided by the `getrusage` system call
  - Not all systems provide all the possible information
  - Hence the `0outputs` result on my system

time()

- Wallclock time is measured by Perl's built-in `time()` function:

```
use LWP::Simple 'get';
my $url= shift;
my $start = time();
my $doc = get($url);
my $elapsed = time() - $start;
print "$elapsed second(s) elapsed.\n";
```

```
2 second(s) elapsed.
```

- It returns the amount of time that has elapsed since the beginning of 1970
- By default, the resolution of `time` is only one second
- Related: `$^T` variable contains the time at which the program started

```
print "Program has been running for ",
           time() - $^T, " second(s).\n";
```

`Time::HiRes`

- Since 5.7.2, Perl has come with the `Time::HiRes` module
  - `Time::HiRes` overloads `time` and `sleep` to have finer resolution

```
use LWP::Simple 'get';
use Time::HiRes 'time';
my $url= shift;
my $start = time();
my $doc = get($url);
my $elapsed = time() - $start;
print "$elapsed second(s) elapsed.\n";
```

```
1.49982798099518 second(s) elapsed.
```

- `Time::HiRes` is also available from CPAN
- It also provides high-resolution versions of `sleep` and `alarm`
  - Also other high-resolution time-related functions

times()

- CPU time is measured with the built-in `times()` function

```
($u, $s, $cu, $cs) = times();
```

- `$u` and `$s` are the user and system CPU times consumed by this process
- `$cu` and `$cs` are the CPU time consumed by descendant processes of this one
  - (These are unavailable on Windows systems)

```
# busyloop
use Time::HiRes 'time';
($parent_run, $child_run) = @ARGV;
$start = time;
until (time >= $start + $parent_run) {
  # Busy loop
}
if (fork) {                         # parent
  wait;
} else {                            # child
  $start = time;
  until (time >= $start + $child_run) {
    # Busy loop
  }
  exit;
}
printf (<<EOF, times());
 u: %.2f  s: %.2f
cu: %.2f cs: %.2f
EOF

% ./busyloop 6 2
 u: 5.11  s: 0.88
cu: 1.61 cs: 0.40
```

- Most benchmarking tools are based on `times`

Simple Benchmarker

```
substr($s, 0, 3) = "abc"

$s =~ s/.../abc/s
```

- Which is faster?

```
my $N = shift || 1000000;
my $s = shift || "The quick brown fox jumps over the lazy dog";

my ($su, $ss) = times;
for (1 .. $N) { substr($s, 0, 3) = "abc" }
my ($eu, $es) = times;
my ($tu, $ts) = ($eu - $su, $es - $ss);
my $total = $tu + $ts;
printf "%20s %5.2f %5.2f %6.2f\n", "substr", $tu, $ts, $total;

my ($su, $ss) = times;
for (1 .. $N) { $s =~ s/.../abc/s}
my ($eu, $es) = times;
my ($tu, $ts) = ($eu - $su, $es - $ss);
my $total = $tu + $ts;
printf "%20s %5.2f %5.2f %6.2f\n", "regex", $tu, $ts, $total;
```

- The output:

```
        substr  5.04  0.01   5.05
         regex  5.71  0.00   5.71
```

Simple Benchmarker

```
              substr  5.04  0.01   5.05
               regex  5.71  0.00   5.71
```

- Looking at this output, we might conclude that the `substr` was 11.5% faster than the regex

- But something important is missing from this output

- The benchmark apparatus itself is biasing the results

```
my ($su, $ss) = times;
for (1 .. $N) { }
my ($eu, $es) = times;
my ($tu, $ts) = ($eu - $su, $es - $ss);
my $total = $tu + $ts;
printf "%20s %5.2f %5.2f %6.2f\n", "NULL", $tu, $ts, $total;
```

- Now the output is:

```
                NULL  1.24  0.00   1.24
              substr  5.10  0.01   5.11
               regex  5.69  0.00   5.69
```

- The time actually spent doing `substr` was about 3.87 seconds

- The time actually spent doing `regex` was about 4.45 seconds

- The `substr` is actually more like 13% faster

Benchmark.pm

- Perl comes with a benchmarking module called `Benchmark`
- The previous slide's benchmark looks like this:

```
use Benchmark;
my $N = shift || 1000000;
my $s = shift || "The quick brown fox jumps over the lazy dog";
timethese($N,
          { substr => sub { substr($s, 0, 3) = "abc" },
            regex  => sub { $s =~ s/.../abc/s},
          });

   regex:  7 wallclock secs
      ( 7.85 usr +  0.00 sys =  7.85 CPU)
      @ 127388.54/s (n=1000000)
   substr:  7 wallclock secs
      ( 8.24 usr +  0.00 sys =  8.24 CPU)
      @ 121359.22/s (n=1000000)
```

- `Benchmark` says that the regex is about 5% faster
    - It tries to do its own adjustments for error

Benchmark.pm

- I don't use `Benchmark.pm` any more
- That's for several reasons
- Here's the results of five consecutive runs of the same benchmark

```
 regex:  ( 7.79 usr +  0.01 sys =  7.80 CPU)
substr:  ( 7.34 usr +  0.02 sys =  7.36 CPU)

 regex:  ( 8.02 usr +  0.00 sys =  8.02 CPU)
substr:  ( 7.04 usr +  0.00 sys =  7.04 CPU)

 regex:  ( 7.95 usr +  0.01 sys =  7.96 CPU)
substr:  ( 7.63 usr +  0.00 sys =  7.63 CPU)

 regex:  ( 8.28 usr +  0.01 sys =  8.29 CPU)
substr:  ( 7.40 usr + -0.01 sys =  7.39 CPU)

 regex:  ( 8.04 usr + -0.03 sys =  8.01 CPU)
substr:  ( 6.92 usr +  0.00 sys =  6.92 CPU)
```

- Problem #1: The individual measurements vary by up to 7%
- Problem #2: Some of the tests are running backwards in time
  - I've also seen:

```
null: -1 wallclock secs (-0.07 usr +  0.01 sys = -0.06 CPU)
  @ -16666666.67/s (n=1000000)
```

Benchmark.pm

- Problem #3:

```
                  regex             substr
Benchmark.pm      8.01              7.36
Handwritten       5.69              5.11
```

- Which one is closer to the truth?
- Here are five consecutive runs of the handwritten benchmark:

```
  NULL  1.23  0.00   1.23
substr  5.07  0.00   5.07
 regex  5.71  0.00   5.71

  NULL  1.24  0.00   1.24
substr  5.07  0.00   5.07
 regex  5.69  0.00   5.69

  NULL  1.23  0.00   1.23
substr  5.07  0.00   5.07
 regex  5.71  0.00   5.71

  NULL  1.23  0.00   1.23
substr  5.07  0.00   5.07
 regex  5.69  0.00   5.69

  NULL  1.25  0.00   1.25
substr  5.05  0.00   5.05
 regex  5.68  0.00   5.68
```

- Here the variation is less than 1%
- I find that I believe these results more than `Benchmark`'s

The Uncertainty Principle

- Heisenberg said that it’s impossible to measure something without altering the measurement
- That is certainly true of benchmarking
- Every benchmark introduces some bias into the thing it purports to measure
- You can try to minimize this in at least two ways
  - One way is to make the benchmark apparatus as simple and as lightweight as possible
  - Then the effects will be small
  - Or, if not, it will be clear what the biases might be

The Uncertainty Principle

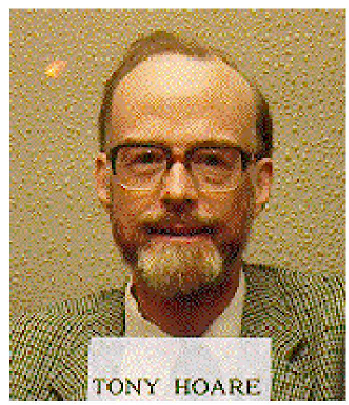
- There's another way to try to eliminate bias
  - You can try to correct for it
  - By adding a lot of complicated machinery to measure bias and subtract it from the results
- This is the `Benchmark.pm` approach
- But if it goes wrong, you have no idea what really happened

```
null: -1 wallclock secs
  (-0.07 usr +  0.01 sys = -0.06 CPU)
  @ -16666666.67/s (n=1000000)
```

- Even when it goes right, you have no idea what really happened

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies."

-- C. A. R. Hoare



- These days I always write my benchmarks manually
- Or I have `Benchmark::Accurate` write the script for me

Performance Tuning Plan

- A program is taking too long to run
- We want to speed it up
- First figure out if it is CPU-bound, memory-bound, or I/O bound
  - Or possibly some of each
- If CPU-bound, use a *profiler* to find CPU-bound parts of the program
  - Then think hard about just those parts
- Come up with a plausible improvement
  - **Test** the 'improved' version to make sure it does the same thing
  - Time the 'improved' version against the original
  - If the new version is faster, weigh the benefit against the costs
    - For example, is the code more complicated now?
    - If so, is it worth it?
- Throughout, try to estimate whether it wouldn't be cheaper in the long run to just buy more hardware

Profilers

- A *profiler* divides the program into small chunks (lines or subroutines)
  - It reports the time taken by each chunk
  - It tells you which chunks contribute the most run time
- Why is this important?
- Suppose you have a program that needs to run as fast as possible
  - You say "Aha! The keyword search function is too slow. I will speed it up."
  - You get out the benchmarker and get to work
  - You research more efficient algorithms
  - You try many different keyword search strategies

Profilers

- All this hard work pays off!
  - Two weeks later the keyword search is twice as fast
- But it turns out that the program was spending only 2% of its time doing keyword search
  - So now it is spending only 1% of its time doing keyword search
- Two weeks down the drain
- This happens to people all the time
- Don’t let it happen to you

Profilers

- The profiler will tell you which parts of the program contribute most of the run time
- This, in turn, allows you to identify the likely targets for improvement

Sample Program: Mail Folder Analyzer

- I wanted a sample program written by someone else
- This one was kindly provided by Mr. Robert Spier
  - He used it in *last* year's optimization tutorial
- It's in `mfa1-n.pl`
- The program analyzes an `mbox`-format file

```
perl mfa1.pl MBOX
```

- The output might look like this:

```
Messages : 109
Total Size : 190790
Average Size : 1750
Most Common Characters:
  : 25557
e : 13719
o : 9330
t : 7473
r : 7460
Least Common Characters:
~ : 18
# : 14
\ : 9
& : 6
Z : 2
| : 2
Most Common Domains:
plover.com : 52
upenn.edu : 38
pobox.com : 19
```

Sample Program: Mail Folder Analyzer

- Timing:

```
real    0m8.356s
user    0m6.770s
sys     0m0.030s
```

- Let's see what we can do about that
- Perl comes standard with a module called `Devel::DProf`
- This module records subroutine entry and exit times as the program runs
- It leaves behind this trace data in a file called `tmon.out`
- To use it:

```
perl -d:DProf mfa1.pl MBOX  > /dev/null
```

- Send output to `/dev/null` to avoid device-related biases

Devel::DProf

- To analyze the `tmon.out` file, you run `dprofpp`
- It gets a lot of options to control the format of the report it generates
- By default it looks like this:

```
Total Elapsed Time = 7.592672 Seconds
  User+System Time = 7.122672 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 30.1   2.149  2.125   4104   0.0005 0.0005  Mail::Header::_fold_line
 24.6   1.756  3.414   2052   0.0009 0.0017  Mail::Header::_fmt_line
 12.2   0.870  0.869    109   0.0080 0.0080  main::letter_histogram
 6.60   0.470  0.458   2052   0.0002 0.0002  Mail::Header::_insert
 5.77   0.411  4.275    109   0.0038 0.0392  Mail::Header::extract
 5.34   0.380  0.367   2161   0.0002 0.0002  Mail::Header::_tag_case
 5.34   0.380  0.358   3604   0.0001 0.0001  Mail::Header::fold_length
 3.72   0.265  1.377    109   0.0024 0.0126  Mail::Header::fold
 2.39   0.170  0.170      1   0.1700 0.1700  Mail::Util::read_mbox
 1.40   0.100  5.917    116   0.0009 0.0510  Mail::Internet::BEGIN
 1.40   0.100  0.269      4   0.0250 0.0674  main::BEGIN
 0.70   0.050  0.048    327   0.0002 0.0001  Mail::Internet::body
 0.70   0.050  5.700    109   0.0005 0.0523  Mail::Header::header
 0.56   0.040  0.091    218   0.0002 0.0004  Mail::Internet::as_string
 0.42   0.030  0.030      1   0.0300 0.0300  warnings::BEGIN
```

- This lists the 15 subroutines that consumed the most total CPU time
- The top 5 account for 80% of the program’s run time

The 90-10 Rule

- The 90-10 rule says that 10% of the code accounts for 90% of the run time
- The other 90% of the code is:
  - Special cases (executed infrequently)
  - Initialization code (executed only once per run)
  - Error handlers (executed never)
- More conservative version: The 80-20 rule
- I counted the lines to see if this was true
  - If anything, ’90-10’ may be too conservative
  - See the Bonus Slides for details

Devel::DProf

```
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 30.1   2.149  2.125   4104   0.0005 0.0005  Mail::Header::_fold_line
 24.6   1.756  3.414   2052   0.0009 0.0017  Mail::Header::_fmt_line
 12.2   0.870  0.869    109   0.0080 0.0080  main::letter_histogram
 6.60   0.470  0.458   2052   0.0002 0.0002  Mail::Header::_insert
 5.77   0.411  4.275    109   0.0038 0.0392  Mail::Header::extract
 ...
```

- About 30% of the program's total run time was spent inside `Mail::Header::_fold_line`
  - Another 24% was spent in `Mail::Header::_fmt_line`
- 8 of the top 15 functions, totaling 82% of the run time, are in `Mail::Header`
- Tentative conclusion: To make this program faster, get rid of `Mail::Header`

Mail::Header

- `Mail::Header` is loaded by `Mail::Internet`
- Let's see where `Mail::Internet` is used:

```
sub handle_message {
  my $message = $_[0];
  my $mi = Mail::Internet->new($message);

  $count++;
  $total_size += length $mi->as_string;
  letter_histogram( $mi->as_string );
  from_histogram( $mi->head->get("From:") );
}
```

- It would appear that it is being used to:

  1. Convert the message to an object and then back to a string, and
  2. to extract the `From` header

handle_message

- Let's try doing those things manually instead

```
sub handle_message {
  my $message = join "", @{$_[0]};
  my $frompat = qr/^From:\s+.*\n        # initial line
                    (?:\s+.*\n)*      # continuation lines
                /xim;

  $count++;
  $total_size += length $message;
  letter_histogram( $message );
  from_histogram( $message =~ /($frompat)/);
}
```

- The results:

```
Before                  After

real    0m8.356s        real    0m1.259s
user    0m6.770s        user    0m1.230s
sys     0m0.030s        sys     0m0.020s
```

- Well how about that?
    - An 81% speedup

Differences

- When optimizing a program, it's vitally important that you not break it
- Unless you live on the planet where it's important to get the wrong answer as quickly as possible
- So here's what I did:

```
% perl mfa1.pl MBOX > out1
% perl mfa2.pl MBOX > out2
% diff -u out?
```

- We hope that the outputs will be identical
  - If not, we have to worry

Differences

```
% sdiff -w60 out?
```

- Here's the output:

```
Messages : 109                  Messages : 109
Total Size : 190790           | Total Size : 190342
Average Size : 1750           | Average Size : 1746
Most Common Characters:         Most Common Characters:
  : 25557                     |   : 24981
e : 13719                       e : 13719
o : 9330                        o : 9330
t : 7473                      | t : 7515
r : 7460                      | r : 7501
Least Common Characters:        Least Common Characters:
~ : 18                          ~ : 18
# : 14                          # : 14
\ : 9                           \ : 9
& : 6                           & : 6
Z : 2                           Z : 2
| : 2                           | : 2
Most Common Domains:            Most Common Domains:
plover.com : 52                 plover.com : 52
upenn.edu : 38                  upenn.edu : 38
pobox.com : 19                  pobox.com : 19
```

- Uh oh

Differences

```
Total Size : 190790              | Total Size : 190342
Average Size : 1750              | Average Size : 1746
```

- Fortunately, this problem is easy to resolve
- Either the total size was 190342, or it wasn’t

```
% wc -c MBOX
 190342 MBOX
```

- How about that?
  - Optimizing the program fixed a bug
- Running the messages through `Mail::Internet->new->as_string` altered them
  - Trailing spaces were trimmed from some header lines
  - The continuation characters were changed in other headers
  - Capitalization was changed in some header field names

```
In-Reply-To: ...                 In-reply-to: ...

t : 7473                         | t : 7515
r : 7460                         | r : 7501
```

- All this will alter the character counts

Mail Folder Analyzer Revisited

- Back to the MFA
- The profiler says that `main::letter_histogram` is consuming most of the CPU time

```
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 63.7   0.830  0.829    109   0.0076 0.0076  main::letter_histogram
 13.8   0.180  0.180      1   0.1800 0.1800  Mail::Util::read_mbox
```

- A 20% speedup in this one function would reduce the program's run time by 1/8

```
sub letter_histogram {
  my $strdex = (length $_[0])-1;
  $letter_hist{substr($_[0],$_,1)}++     for (0..$strdex);
}
```

- Not much to work with here
  - I tried a bunch of things I thought of and some the test audiences suggested
  - No luck
  - Some of these things are in the Bonus Section at the end

Mail Folder Analyzer Revisited

- We couldn't get any improvement from `letter_histogram`

```
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 63.7   0.830  0.829    109   0.0076 0.0076  main::letter_histogram
 13.8   0.180  0.180      1   0.1800 0.1800  Mail::Util::read_mbox
 4.61   0.060  0.130      3   0.0200 0.0432  main::BEGIN
 4.61   0.060  0.887    109   0.0005 0.0081  main::handle_message
```

- Maybe look into `read_mbox` now?

- No.
- The profiler is telling us something extremely important here:
  - Trying to speed up the program any more would be a **waste of effort**
- The next biggest target is `Mail::Util::read_mbox`
  - But a 20% speedup here would only get us a 2.8 % overall speedup
  - That's a total of about 36 milliseconds per run
  - Would it really be worth the trouble?

When It’s Time to Give Up

```
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 63.7   0.830  0.829    109   0.0076 0.0076  main::letter_histogram
 13.8   0.180  0.180      1   0.1800 0.1800  Mail::Util::read_mbox
 4.61   0.060  0.130      3   0.0200 0.0432  main::BEGIN
 4.61   0.060  0.887    109   0.0005 0.0081  main::handle_message
```

- We could conceivably save up to 180 ms per run by sufficiently clever hacking of `read_mbox`
    - How much is that pony really worth?
- Say my computer cost $3000 and has a lifetime of about 5 years
    - That’s about .0019 cents per CPU-second
    - The benefit of a 20% speedup in `read_mbox` is about .000000676 dollars per run
    - That’s the pony. What is the price?
- My time bills at a fairly high rate, but let’s say it’s $50 per hour
    - I might spend 20 minutes getting the speedup
- To break even, I would have to run the program about 25 million times
- Of course, this is much more likely if the program has 25 million users

The Big Picture

- People waste a huge amount of time on performance improvements
- Here's a more common situation
  - A programmer is assigned to make program *X* faster
  - The programmer spends a week on the project
  - The programmer's salary is US$65,000 per year
  - Cost of project: $2,600 (counting overhead, benefits, etc.)
- Compare this cost with the cost of buying another Gb of memory
  - Or a really hot CPU upgrade
  - Or a second server
- Often, the hardware purchase is a lot more cost-effective
- It is also more likely to be successful

The Big Picture

- Through the 1960s, hardware was terribly expensive
- Machines were physically large and computationally small

"The late Professor Don Gillies at Illinois claimed to have written the first assembler. . . .

"Gillies was a grad student of John Von Neumann, working on the IAS machine at Princeton. He was supposed to be working as a coder, translating programs written by more advanced researchers into machine code, but he found the job tedious, and wrote an assembler to help him do it faster.

"John Von Neumann's reaction was extremely negative. Gillies quotes his boss as having said 'We do not use a valuable scientific computing instrument to do clerical work!'"

- (This was reported by Doug Jones of U. Iowa; Gillies was his thesis advisor)
    - (If true, it would have taken place around 1953)
- The discipline of computer programming was forged in this environment
- It gave us a hangover
- We still think like this

POD Formatting

- I use the documentation all the time

```
% time perldoc perlfunc > /dev/null

real    0m24.396s
user    0m22.170s
sys     0m0.370s
```

- But I'd use it more if `perldoc` weren't so slow
- This section is about the `perldoc` that comes with Perl 5.8
- Perl documentation comes in the very simple POD format
  - `pod2man` translates POD to the Unix man page format
  - `nroff` formats man pages for display on a terminal

POD Formatting

- First, a note about The Big Picture
- If `perldoc` is slow, the best solution might not to be to speed it up
- The best solution might be more like this:

```
for i in /src/perl-5.8.0/pod/*; do
  j=‘basename $i .pod‘
  pod2man $i > /usr/local/man/man1p/$j.1p
  man -F $j
done
```

- Then you can use `man perlfunc` or whatever
- Perl does this automatically when it is installed
- Still, there is some value in speeding up `perldoc`
- Installing the Perl/Tk documentation takes a very long time

perldoc

- `perldoc` is mostly just a wrapper around `pod2man`
- It locates files and invokes `pod2man` and `nroff` as necessary
- Let's find out how to run `pod2man`:

```
% strace -s10000 -f -o perldoc.trace perldoc perlfunc > /dev/null
```

- This generates a list of every system call run by `perldoc`
  - In particular, it will tell us what commands `perldoc` ran

```
% grep execve perldoc.trace
21892 execve("/usr/local/bin/perldoc", ["perldoc", "perlfunc"], ...
21893 execve("/bin/sh", ["sh", "-c",
   "/usr/local/bin/pod2man --lax /usr/local/lib/perl5/5.8.0/pod/perlfunc.pod | nroff -man"], ...
21894 execve("/usr/local/bin/pod2man",
   ["/usr/local/bin/pod2man", "--lax",
    "/usr/local/lib/perl5/5.8.0/pod/perlfunc.pod"], ...
21895 execve("/usr/bin/nroff", ["nroff", "-man"], ...
```

- Now let's run `pod2man` the same way:

```
%  time /usr/local/bin/pod2man --lax
     /usr/local/lib/perl5/5.8.0/pod/perlfunc.pod > /dev/null

real    0m18.158s
user    0m17.670s
sys     0m0.080s
```

- Yup
  - Probably a lot of the rest is in `nroff`
  - Presumably we're not prepared to do anything about `nroff`

pod2man

- Now we pull out the profiler:

```
% perl -d:DProf  ./pod2man-1.pl --lax < perlfunc.pod
    > perlfunc.man
```

- Save the output so that we can check future outputs against it

```
% dprofpp tmon.out > dp.out
```

```
Total Elapsed Time = 21.27246 Seconds
  User+System Time = 19.99246 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 23.4   4.686 14.255   1440   0.0033 0.0099  Pod::Parser::parse_text
 9.55   1.909  1.887   3609   0.0005 0.0005  Pod::Man::guesswork
 6.54   1.307 20.109      1   1.3073 20.109  Pod::Parser::parse_from_filehandle
 5.99   1.197 18.354   1609   0.0007 0.0114  Pod::Parser::parse_paragraph
 5.95   1.189  1.140   7733   0.0002 0.0001  Pod::ParseTree::append
 5.79   1.158  1.349   2121   0.0005 0.0006  Pod::InteriorSequence::new
 4.68   0.936  3.004   3561   0.0003 0.0008  Pod::Man::collapse
 3.99   0.797  2.547   2121   0.0004 0.0012  Pod::Man::sequence
 3.50   0.700  0.692   1208   0.0006 0.0006  Pod::Man::textmapfonts
 2.80   0.559  0.600   3561   0.0002 0.0002  Pod::ParseTree::_unset_child2paren
```

- Clearly `parse_text` is the big target here

`parse_text`

- `parse_text` is about 76 lines long
- Its job is to take apart a POD paragraph like this:

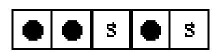
```
Be aware that the optimizer might have optimized call frames
away before C<caller> had a chance to get the information.
That means that C<caller(N)> might not return information
about the call frame you expect it do, for C<< N > 1 >>.  In
particular, C<@DB::args> might have information from the
previous time C<caller> was called.
```

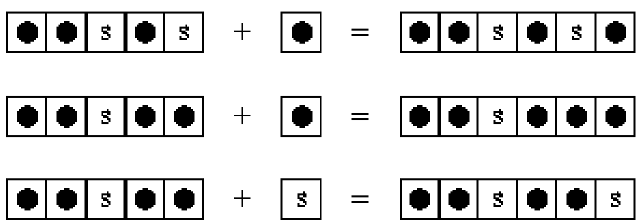
- Locate the escaped sections like `C<caller>` and `C<< N > 1 >>`
- Next step: Grovel over `parse_text` until you understand it

Pod::ParseTree::append

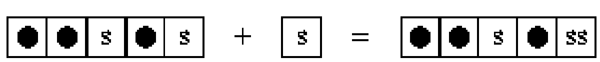
- Digression: While grovelling over the POD parser code, I wandered in here:

```
sub append {
   my $self = shift;
   local *ptree = $self;
   for (@_) {
      next  unless length;
      if (@ptree  and  !(ref $ptree[-1])  and  !(ref $_)) {
         $ptree[-1] .= $_;
      }
      else {
         push @ptree, $_;
      }
   }
}
```

- A `Pod::ParseTree` object is basically an array of strings and objects 
- Normally, we can use `push` to append a new item to the array



- But if the last element of the array is a string,
  - and the thing we're appending is also a string
  - then we can concatenate the two strings instead



Pod::ParseTree::append

- I wondered if it was possible to simplify this
- What's the next thing I did?
- I checked to `dprofpp` output to see if `append` was worth investigating

```
5.95   1.189  1.140   7733   0.0002 0.0001  Pod::ParseTree::append
```

- It's tied for fourth place
    - It's also small
- It should be worth a little effort

Pod::ParseTree::append

```
sub append {
   my $self = shift;
   local *ptree = $self;
   for (@_) {
      next  unless length;
      if (@ptree  and  !(ref $ptree[-1])  and  !(ref $_)) {
         $ptree[-1] .= $_;
      }
      else {
         push @ptree, $_;
      }
   }
}
```

- What if we didn't bother to agglomerate strings?
- Then `append` would become:

```
sub append {
  my $self = shift;
  push @$self, @_;
}
```

- It's easy to imagine that this would speed up `append` substantially

Pod::ParseTree::append

- Will failing to agglomerate strings cause any problems?
- There might be code that is depending on there not being two consecutive strings
- But I don't think there is
- Access to `Pod::ParseTree` objects is mediated by methods like this:

```
sub raw_text {
   my $self = shift;
   my $text = "";
   for ( @$self ) {
      $text .= (ref $_) ? $_->raw_text : $_;
   }
   return $text;
}
```

- This will work fine if I change `append`
- Let's give it a try

`Pod::ParseTree::append`

- To test my change, I created a local `Pod` directory
  - Copied `Pod/InputObjects.pm` into it
  - Modified my `Pod/InputObjects.pm`
  - Then ran:

```
% perl -I. `which pod2man` < perlfunc.pod > append-after.out
```

- Preliminary results:
  - Correctness:

```
% cmp perlfunc.man append-after.out
```

- Timing:

```
        Before                      After

        real    0m26.232s           real    0m22.225s
        user    0m24.520s           user    0m20.870s
        sys     0m0.420s            sys     0m0.490s
```

- I also reran the `Pod::` test suite to make sure I didn't break anything
- End of digression

parse_text

- The next thing that occurs to me: parse_text is complicated because of C<< a->b >> and such
  - There's a lot of parsing
  - And a delimiter stack in case of A<< foo B<<< c->d >>> bar >>
  - And a lot of special-casery
  - But these complicated cases rarely if ever come up
- The common case is very simple
  - Typically, something like C<caller>

Optimize for the common case.

- Doing this is a rather involved exercise in maintenance programming
  - I love maintenance programming

`parse_text`

- `parse_text` splits the input into a list of *tokens*
- Then it deals with the tokens one at a time
- The existing tokenizer splits `C<caller>` into two tokens:
  - `C<` and `caller>`
  - It puts an object representing `C<` onto the stack
  - Then when it sees `caller>` it pops the stack
- This complication is necessary for difficult cases like `A<foo B<bar> baz>`
- For simple cases it is overkill
- Idea:
  - Tokenize difficult cases as before
  - But tokenize simple cases like `C<caller>` as single tokens

parse_text

- At this point I built a test case

```
This is a small stress test of the B<pod delimiter> mechanism.  You
are allowed to have X<< double >> and even Y<<< triple >>> delimiters.
Ordinary Z<single I<delimiters> may be> nested or may contain A<funny
< characters>.  C<< Double D<delimiters> may >> E<< also F<<< nest >>>
if desired >>.
```

- Old tokenization:

```
(This is a small stress test of the )
(B<)
(pod delimiter> mechanism.  You\nare allowed to have )
(X<< )
(double >> and even )
(Y<<< )
(triple >>> delimiters.\nOrdinary )
(Z<)
(single )
(I<)
(delimiters> may be> nested or may contain )
(A<)
(funny \n< characters>.  )
(C<< )
(Double )
(D<)
(delimiters> may >> )
(E<< )
(also )
(F<<< )
(nest >>>\nif desired >>.\n)
```

`parse_text`

- New tokenization:

```
(This is a small stress test of the )
(B<pod delimiter>)
( mechanism.  You\nare allowed to have )
(X<< )
(double >> and even )
(Y<<< )
(triple >>> delimiters.\nOrdinary )
(Z<)
(single )
(I<delimiters>)
( may be> nested or may contain )
(A<)
(funny\n< characters>.  )
(C<< )
(Double )
(D<delimiters>)
( may >> )
(E<< )
(also )
(F<<< )
(nest >>>\nif desired >>.\n)
```

- So we now need to add handlers for the new `X<complete sequence>` tokens
- In the old regime, the sequence would be put on the stack, then taken off again
    - We'll just do that in one fell swoop

parse_text

- Old tokenizer code:

```
split /([A-Z]  <              # Escape code and open bracket
        (?: <+ \s ) ?         # Possible extended delimiter
       )/x;
```

- New tokenizer:

```
split /([A-Z]  <              # Escape code and open bracket
        (?: [^<>]* >          # ...and the rest of the escape sequenc
          | (?: <+ \s )?      # OR a possible extended delimiter
        ))/x;
```

parse_text

- Old code:

```
elsif ( /^([A-Z])(<(?:<+\s)?)$/ ) {
  ## Push a new sequence onto the stack of those "in-progress"
  ($cmd, $ldelim) = ($1, $2);
  $seq = Pod::InteriorSequence->new(
             -name   => $cmd,
             -ldelim => $ldelim,  -rdelim => '',
             -file   => $file,    -line   => $line
         );
  $ldelim =~ s/\s+$//, ($rdelim = $ldelim) =~ tr/</>/;
  (@seq_stack > 1)  and  $seq->nested($seq_stack[-1]);
  push @seq_stack, $seq;
}
```

- This handles the `X<` part of a sequence
- It builds a new `Pod::InteriorSequence` and puts it on the stack
- Later code takes the remainder, `complete sequence> blah blah`
  - Expands `complete sequence` if necessary
  - Appends it to the `Pod::InteriorSequence` object
  - Puts `blah blah` back into the input stream
- There's a lot of state variable management and stack jiggery-pokery

`parse_text`

- My first cut at a special case for `C<simple>` was:

```
## Look for an entire simple sequence 20030420 mjd@plover.com
if ( /^([A-Z])<([^<>]*)>$/ ) {
    $seq = Pod::InteriorSequence->new(
               -name   => $1,
               -ldelim => "<",  -rdelim => ">",
               -file   => $file,    -line   => $line
           );
    $seq->append($2);
    $seq_stack[-1]->append($expand_seq
                            ? &$xseq_sub($self, $seq)
                            : $seq);
}
... the rest as before ...
```

- I just cribbed most of this from further down
- I chopped out the parts that seemed unnecessary
- Filled in `-rdelim` since it was known immediately
- The `->append($2)` code is simple because I know that `$2` is a plain string
    - (The original version was more like the second `append` call)
- I don't have to put `C<...` on the stack while I go looking for `...>`.

parse_text

- Then I ran the tests
- They *almost* all passed

```
basic...........ok 2/11Can't call method "raw_text"
   on unblessed reference at ../Pod/InputObjects.pm line 618,
   <GEN0> line 129.
basic...........dubious
        Test returned status 255 (wstat 65280, 0xff00)
DIED. FAILED tests 3-11
        Failed 9/11 tests, 18.18% okay
```

- Not bad considering I don't know what I am doing
- I will spare you the details of the next 90 minutes of debugging
- The answer: I missed copying one of the lines from the other blocks!

```
if ( /^([A-Z])<([^<>]*)>$/ ) {
    $seq = Pod::InteriorSequence->new(
               -name   => $1,
               -ldelim => "<",  -rdelim => ">",
               -file   => $file,    -line   => $line
           );
    $seq->append($2);
    $seq->nested($seq_stack[-1]) if @seq_stack > 1;
    $seq_stack[-1]->append($expand_seq
                           ? &$xseq_sub($self, $seq)
                           : $seq);
}
```

- Whoops!

The Moment of Truth

```
Before                      After

real     0m26.957s          real     0m27.117s
user     0m24.180s          user     0m22.020s
sys      0m0.550s           sys      0m0.480s
```

- Not bad for one change (about 9%)
- The outputs are identical
    - Before:

```
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 22.9   4.507 14.205   1440   0.0031 0.0099  Pod::Parser::parse_text
```

- After:

```
 19.4   3.515 12.303   1440   0.0024 0.0085  Pod::Parser::parse_text
```

`Devel::SmallProf`

- Another useful tool for profiling is `Devel::SmallProf`
- Instead of measuring the contribution per subroutine, it measures contribution per line
- Of course, it is even less accurate than `Devel::DProf`
- It's available on CPAN, but isn't standard
- To use it:

```
% perl -d:SmallProf  ./pod2man-1.pl --lax ...
```

- It leaves behind a report in `smallprof.out`

smallprof.out

```
================ SmallProf version 0.9 ================
Profile of Pod/Parser.pm                          Page 174
```

```
     count wall tm  cpu time line
        0 0.000000 0.000000   785:    ## capturing parens keeps the delimiters)
     1440 0.175561 0.200000   786:    $_ = $text;
        0 0.000000 0.000000   787:#    my @tokens = split /([A-Z]<(?:<+\s)?)/;
     1440 0.286681 0.460000   788:    my @tokens = split /([A-Z]  <          #
        0 0.000000 0.000000   789:                (?: [^<>]* >      # ... and
        0 0.000000 0.000000   790:                  | (?: <+ \s )?  # OR a
        0 0.000000 0.000000   791:                    ))/x;
        0 0.000000 0.000000   792:#    { local $" = ")\n("; warn "tokens:
     7160 0.561213 0.970000   793:    while ( @tokens ) {
     5720 0.523698 0.900000   794:        $_ = shift @tokens;
     5720 0.376381 0.770000   795:        next unless length;
        0 0.000000 0.000000   796:        ## Look for an entire simple sequence
     5652 0.924686 1.030000   797:        if ( /^([A-Z])<([^<>]*)>$/ ) {
     2083 1.415592 1.390000   798:            $seq = Pod::InteriorSequence-
        0 0.000000 0.000000   799:                       -name   => $1,
        0 0.000000 0.000000   800:                       -ldelim => "<",   -
        0 0.000000 0.000000   801:                       -file   => $file,    -
        0 0.000000 0.000000   802:                   );
     2083 1.182618 1.080000   803:            $seq->append($2) if length($2);
     2083 0.179391 0.220000   804:            $seq->nested($seq_stack[-1]) if
     2083 0.576378 0.540000   805:            $seq_stack[-1]-
        0 0.000000 0.000000   806:        }
        0 0.000000 0.000000   807:        ## Look for the beginning of a
        0 0.000000 0.000000   808:        elsif ( /^([A-Z])(<(?:<+\s)?)$/ ) {
        0 0.000000 0.000000   809:            ## Push a new sequence onto the
       38 0.003812 0.010000   810:            ($cmd, $ldelim) = ($1, $2);
       38 0.029990 0.020000   811:            $seq = Pod::InteriorSequence-
        0 0.000000 0.000000   812:                       -name   => $cmd,
        0 0.000000 0.000000   813:                       -ldelim => $ldelim,  -
        0 0.000000 0.000000   814:                       -file   => $file,    -
        0 0.000000 0.000000   815:                   );
```

`smallprof.out`

- To do anything useful with this, we’d have to extract the section of interest
- Then trim out the page headers
- Then sort the lines in ascending order by CPU time
- It’s easier and more useful to replace `Devel::SmallProf`
- You can write your own `Devel::` modules
- They get access to the same debugger hooks that other `Devel::` modules do

Debugger Features

- Lots of functions for haruspication
- See `perldebguts` (or `perldebug`) for fullest details

- `@{"::_<foo.pl"}` contains the source code of `foo.pl`
- `%DB::sub` contains subroutine start-end information
- `DB::DB()` is called before each executed line
- `caller()` returns current package, filename, line as usual
- `caller()` also sets `@DB::args` when called from package `DB`

Trivial Debugger

```
package Devel::Count;

sub DB::DB { ++$count }

END { print STDERR "Total statements: $count\n" }
```

- Now `perl -d:Count anyprogram.pl` prints out:

```
Total statements: 286
```

Devel::OurProf

```
package Devel::OurProf;
BEGIN { ($start_time) = times
        open REPORT, ">", "ourprof.out" or die $! }

sub DB::DB {
  my ($end_time) = times;
  my $elapsed = $end_time - $start_time;
  my ($package, $filename, $line) = caller(0);
  my $sub = (caller(1))[3];
  ($start_time) = times, return
    unless $sub eq ’Pod::Parser::parse_text’;
  $count[$line]++;
  $time[$line] += $elapsed;
  $total_time += $elapsed;
  ($start_time) = times;
}

... continued ...
```

Devel::OurProf

```
END {  # Print out the report
  select REPORT;
  my @r;
  my @line_ranks = sort {$time[$b] <=> $time[$a]} (1 .. $#time);
  @r[@line_ranks] = ((’*’) x 10, (’+’) x 15, (’-’) x 75, (’.’) x
  for (1 .. $#count) {
    my ($c, $t) = ($count[$_], $time[$_]);
    my $L = ${"::_<Pod/Parser.pm"}[$_];
    chomp $L;
    $L = substr($L, 0, 54);
    if ($c) {
        printf "%4d%s%6d %5.2f %5.2f %-54s\n",
          $_, $r[$_] || ’ ’, $c, $t, 100*$t/$total_time, $L;
    } else {
        printf "%4d                    %-54s\n", $_, $L;
    }
  }
}
```

- The `@r` thing is a little tricky, but it’s just a trick
- `$r[$N]` is a `*` just when `$N` is one of the top 10 longest-running lines
  - It is a `+` when `$N` is ranked 11-25
  - It is a `-` when `$N` is ranked 26-100

ourprof.out

- Here's an excerpt:

```
785                            ## capturing parens keeps the delimiters)
786+  1440  0.10  1.87         $_ = $text;
787                        #    my @tokens = split /([A-Z]<(?:<+\s)?)/;
788+  1440  0.09  1.69         my @tokens = split /([A-Z]  <          # Escap
789                                        (?: [^<>]* >      # ... and the re
790                                          | (?: <+ \s )?  # OR a possible
791                                            ))/x;
792                        #    { local $" = ")\n("; warn "tokens: (@tokens)\
793*  7160  0.64 11.99         while ( @tokens ) {
794*  5720  0.22  4.12             $_ = shift @tokens;
795*  5720  0.25  4.68             next unless length;
796                                ## Look for an entire simple sequence 2003
797+  5652  0.14  2.62             if ( /^([A-Z])<([^<>]*)>$/ ) {
798*  2083  0.19  3.56                 $seq = Pod::InteriorSequence->new(
799                                                -name   => $1,
800                                                -ldelim => "<",  -rdelim =>
801                                                -file   => $file,    -line
802                                            );
803+  2083  0.10  1.87                 $seq->append($2) if length($2);
804+  2083  0.10  1.87                 $seq->nested($seq_stack[-1]) if @seq_s
805+  2083  0.10  1.87                 $seq_stack[-1]->append($expand_seq ? &
806                                }
807                                ## Look for the beginning of a sequence
808                                elsif ( /^([A-Z])(<(?:<+\s)?)$/ ) {
809                                    ## Push a new sequence onto the stack
810-    38  0.01  0.19                 ($cmd, $ldelim) = ($1, $2);
811      38  0.00  0.00                 $seq = Pod::InteriorSequence->new(
812                                                -name   => $cmd,
813                                                -ldelim => $ldelim,  -rdeli
814                                                -file   => $file,    -line
815                                            );
```

- Some of this might be suggestive
- For example, we might try to adjust the tokenizer to avoid generating empty tokens
    - This would obviate line 795

Turnaround

- Sometimes the key performance criterion is *responsiveness*
- Time-sharing systems are a lot less efficient than batch systems
  - But batch systems are dead
  - Because everyone hates them
- I had a client with a CGI application
  - Their client (Ford) would hit the CGI application in large bursts
  - Maybe 2000 times over five minutes
  - Then not at all for a long time
  - How to get the application to reply to Ford in a reasonable amount of time?
- The code is about 430 lines, so we'll only see excerpts

Turnaround

- The first thing the program does is recover an XML file from the CGI request:

```
my $xmlpost = CGI::XMLPost->new();
my $xml = $xmlpost->data();
```

- It saves the XML (actually a SOAP request) to two files:

```
open(OUT,">$outfile");
print OUT $xml;
close(OUT);

open(OUT,">>$dailyfile");
print OUT $outfile,":",$xml,"\n";
close(OUT);
```

- Then it reads the XML back in:

```
my $xs1 = XML::Simple->new();
my $doc;
eval { $doc=$xs1->XMLin($outfile, forcearray => ['Change']); };
```

- If all goes well to this point, it returns a success code back to Ford

- After printing the success or failure code, the program opens a database connection

- It extracts information from the SOAP request and adds it to the database

Turnaround

- The primary problem was the sudden burst of requests all at once
- 3000 instances of the program would run in a few minutes
- These 3000 instances all competed for the CPU and the database
- The programmers tried to improve turnaround time this way:

```
FORK: {
   if ( $pid = fork ) {
      # exit parent
      CORE::exit;
      }
   elsif ( defined $pid ) {
      close(STDIN);
      close(STDOUT);
      close(STDERR);
      open(STDOUT,">>/programs/cassens/DC/CO/eHub/FordXML.stdout");
      open(STDERR,">>/programs/cassens/DC/CO/eHub/FordXML.stderr");
      }
```

- This allows the server to respond to the client immediately
  - The child process goes on to talk to the database
- This made the problem worse, not better
  - 6000 processes instead of 3000

Turnaround

- The biggest improvement:
    - The client converted the CGI script into an Apache plugin module
    - No more 3000 processes
- However, I had some recommendations also
- The major one:
    - Commit the XML to a file, check it, return the status code, and exit
- A separate background process can take care of parsing it and updating the database
- The separate process handles one file at a time
- This makes it possible to control the load
    - Only one background process is running at a time
    - It can go to sleep when system load is high, continue when things cool off

Turnaround

- Also some minor recommendations
- Instead of this:

```
eval { $doc=$xs1->XMLin($outfile, forcearray => ['Change']); };
```

- Just use this:

```
eval { $doc=$xs1->XMLin($xml, forcearray => ['Change']); };
```

- The XML is already in memory (we just wrote it out)
  - So why bother to read it back in again?

Turnaround

- Another minor recommendation: Get rid of `CGI::XMLPost`

```
use CGI::XMLPost;
my $xmlpost = CGI::XMLPost->new();
my $xml = $xmlpost->data();
```

- If you look at the `CGI::XMLPost` code, you discover that what it's doing is:

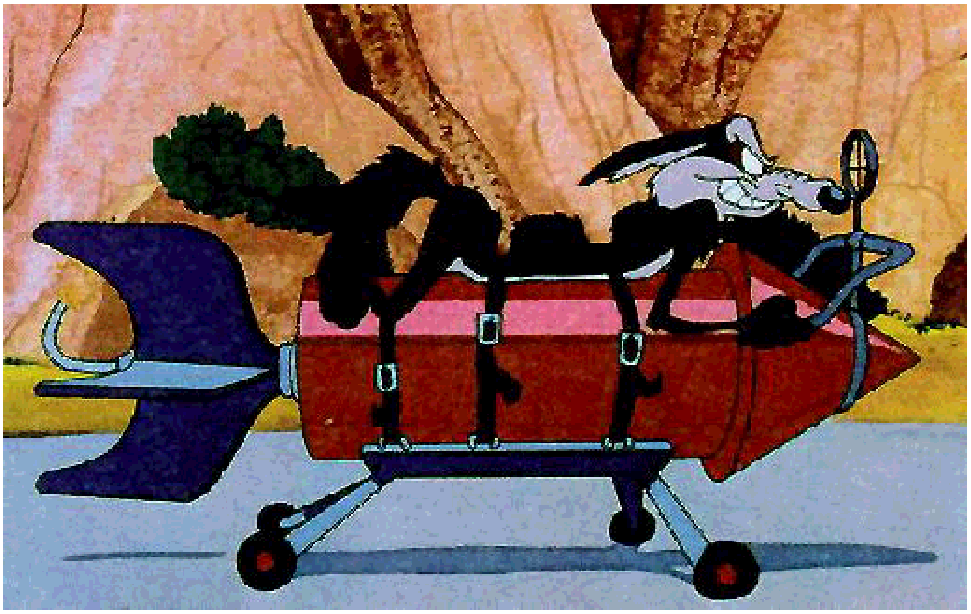
```
my $cl = $ENV{CONTENT_LENGTH};

if ( read( STDIN, $self->{_data}, $cl) == $cl )
{
   return $self;
}
```

- The world is full of useless modules like this
- They exist only to put a hokey OO interface on something that didn't need one
- I suggested replacing it with:

```
my $xml;
my $cl = $ENV{CONTENT_LENGTH};
unless ( read( STDIN, $xml, $cl) == $cl )
{
  print "Status: 404 Not Found\n";
  ...
  print XMLLOG "bad post\n";
  exit;
}
```

Blunders



Pseudo-Hashes

- Hashes are commonly used for objects
  - Keys are member data names, values are member data

```
if ($self->{TYPE} eq 'octopus') {
  $self->{tentacles} = 8;
  $self->{hearts} = 3;
  $self->{favorite_food} = 'crab cakes';
}
```

- But arrays are smaller and faster
- Big disadvantage: Data is referred to by number instead of by name

```
if ($self->[2] eq 'octopus') {
  $self->[17] = 8;
  $self->[4] = 3;
  $self->[28] = 'crab cakes';
}
```

Pseudo-Hashes

- For 5.005, someone had an interesting idea:
- Suppose an object was declared from a certain class, like this:

```
my Critter $self;
```

- Suppose `Critter` objects are based on arrays instead of hashes
- And suppose `Critter.pm` declared its fields at compile time, like this:

```
package Critter;
use fields qw(NAME TYPE size hearts likes_cookies
   ...
        pelagic tentacles is_tasty
   ...
        );
```

- Then when Perl saw `$self->{TYPE}` it could *pretend* you wrote `$self->[2]`
- You would get all the benefits of both!

Pseudo-Hashes

- This idea was developed over the next few years
- Big problem: This cannot be translated at compile time

```
$self->{$key}
```

- Solution: `$self` would be an arrayref that pretended to be a hashref
- It would carry around a hash that mapped keys to values:

```
[ { NAME => 1, TYPE => 2, size => 3, ... },
  "Fenchurch",
  "Octopus", "Small", 3, undef, ... ]
```

- You were now allowed to use an arrayref as if it were a hashref
- This was formerly an error:

```
$array_ref->{$key}
```

- Now it is an abbreviation for this:

```
$array_ref->[$array_ref->[0]->{$key}]
```

- Note that this is somewhat slower than `$hash_ref->{$key}` would have been

Pseudo-Hashes

- It was all very complicated
  - Lots and lots of code had to be added to Perl
  - All sorts of complications
  - `exists` had to be extended to work on arrays
- After it was all done, however, the new improved semantics were 15% faster than the old:

```
    Old                        New

                          package Critter;
                          use fields qw(... hearts ...);

                          my Critter $self;

    $self->{hearts};      $self->{hearts};
```

- So perhaps it was worth all that trouble

The Missing 15%

- A couple of years later, some bright boy finally asked the right question
- He did not compare the new syntax with the old syntax in Perl 5.005
- Instead, he compared the old syntax in 5.005 with the old syntax in 5.004
- 5.005 was 15% slower
- Adding the pseudohash stuff to 5.005 had slowed down **all** hash access by 15%
- In the best possible case, the efficiency gain was just enough to get you back to zero
- Pseudo-hashes are now being withdrawn
- Good riddance

Getting the Wrong Answer as Quickly as Possible

```
Message-ID: <3A317EF2.3000509@klamath.dyndns.org>
Subject: eval() performance
Date: Sat, 09 Dec 2000 00:38:11 GMT

I’ve been taking a look at some old Perl code, written by
someone else.  The main part of the app does the following
(it’s a CGI script):

1 Read in a certain CGI parameter
2 Based on this parameter, open() a certain Perl script as a
text file and read the contents into a single scalar variable
3 Use the following code to evaluate the loaded code:

eval $code;
if ($@) {
     #handle errors
}

My question is: how would you improve this? My first
thought was to use an eval block - i.e.

eval {$code;};
if ($@) {
     #handle errors
}

Would this improve performance?
```

- Good question
- Unfortunately, things started to go awfully wrong at that point

Getting the Wrong Answer as Quickly as Possible

```
>> Would this improve performance?
>
> Write a benchmark and see.

Well alright :-)

#!/usr/bin/perl -w
#test.pl
use strict;
use Benchmark;
undef $/;
my $code;
timethese(8000, {
     'Slow Eval' => sub {open(INPUT, 'code.pl');$code =
<INPUT>;close(INPUT);eval $code;},
     'Fast Eval' => sub {open(INPUT, 'code.pl');$code =
<INPUT>;close(INPUT);eval {$code;};}
});

Results:

Benchmark: timing 8000 iterations of Fast Eval, Slow Eval...
  Fast Eval:  0 wallclock secs
           ( 0.30 usr +  0.13 sys =  0.43 CPU)
  Slow Eval:  6 wallclock secs
           ( 4.98 usr +  0.42 sys =  5.40 CPU)

So apparently an eval block is significantly faster than
calling eval() on a scalar.
```

- Well, that’s good to know
- Anyone see the problem here?

Getting the Wrong Answer as Quickly as Possible

- First, the benchmark code is way too complicated
- I'll use this instead:

```
#!/usr/bin/perl -w
use Benchmark;
my $code = q{"x" . "y"};

timethese(8000, {
     'Slow Eval' => sub {eval  $code  },
     'Fast Eval' => sub {eval {$code} }
});

Benchmark: timing 8000 iterations of Fast Eval, Slow Eval...
 Fast Eval:  0 wallclock secs
        ( 0.02 usr +  0.00 sys =  0.02 CPU)
        @ 400000.00/s (n=8000)
 Slow Eval:  4 wallclock secs
        ( 3.43 usr +  0.00 sys =  3.43 CPU)
        @ 2332.36/s (n=8000)
```

- Looks conclusive, doesn't it?
- Anyone see the problem here?

Getting the Wrong Answer as Quickly as Possible

```
#!/usr/bin/perl -w
use Test::More 'no_plan';
my $code = q{"x" . "y"};
is(eval $code , 'xy', "string eval");
is(eval{$code}, 'xy', "block eval");
```

- Let's make sure those `eval`s are doing what we thought:

```
ok 1 - string eval
not ok 2 - block eval
#     Failed test (evaltest.pl at line 6)
#          got: '"x" . "y"'
#     expected: 'xy'
1..2
# Looks like you failed 1 tests of 2.
```

- How about that
  - The "block `eval`" is not actually eval-ing the code
- `eval {$code}` is not analogous to `eval $code`
  - It is analogous to `eval '$code'`

The Wrong Question


```
"So apparently an eval block is
 significantly faster than
 calling eval() on a scalar."
```

- Yep, benchmarks show that it’s 170 times faster
- But that’s because it doesn’t actually evaluate anything
- Whoops
- If you have code in a string, and you want to execute the code, you *must* use ’string `eval`’
- Asking whether string or block `eval` is faster is The Wrong Question
    - It’s like asking whether a screwdriver is faster than blinking your eyes
    - You can blink your eyes a lot faster than you can use a screwdriver
    - But it won’t help you get that screw in

Trivial Benchmarks

- That's another reason I don't like `Benchmark.pm`
- It makes it too easy to ask the wrong questions
- "Which is faster? Subroutine or method calls?"
- "Which is faster? `map` or `for`?"
- People like to use `Benchmark` to answer questions like this
- But often the best answer is "Who the hell cares?"
- Suppose it turns out that `map` is faster
- Only a pinhead would rewrite all his programs to use `map` instead of `for`
- The difference is going to be minuscule anyway
    - If it isn't, the right response is to file a bug report to p5p

Trivial Benchmarks

```
Newsgroups: comp.lang.perl.misc
Date: Wed, 10 Oct 2001 18:59:17 +0400
Message-ID: <3BC46245.E2B3630A@pisem.net>

Suppose we have $_="haha:lala:rere";
What is faster??
($haha) = split /:/, $_;  # or
($haha) = split(/:/, $_, 1);
```

- Lots of people weighed in on this matter
- Some advised the use of `Benchmark`
- Few noticed that the two samples do not do the same thing
  - Or that the second sample is entirely worthless

```
# $_ = "a:b:c:d"
split /:/, $_, 3;              # ("a", "b", "c:d")
split /:/, $_, 2;              # ("a", "b:c:d")
split /:/, $_, 1;              # ("a:b:c:d")

($haha) = split /:/, $_;       # ("a", "b:c:d")
```

1+1=0

- Consider this:

```
while (<>) {
  my ($n, $text) = split /: /, $_, 2;
  $line[$n] = $text;
}
```

- Each time `$n` is larger than `@line`, the array is extended
    - It might have to be copied to a new, larger region of memory
- Why not extend all at once?
- If you know that `$n` will get as large as 1000000, then:

```
$#line = 1000000;
while (<>) {
  my ($n, $text) = split /: /, $_, 2;
  $line[$n] = $text;
}
```

- This should save time

1+1=0

- One day in 1998 Jon Orwant posted to `perl5-porters`
- He had benchmarked the `$#line = 1000000` optimization
- It was not speeding anything up

I tried to quantify the speedup of preallocating arrays, and found that it actually slows your code down. Always. Several benchmarks on several platforms with several versions of Perl 5 all chanted in unison: Avoid setting `$#array`.

- (`http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/1998-04/msg`
- There was a big hue and cry over this
  - "`$#line = 1000000` must be broken!"

1+1=0

- Here's Jon's benchmark:

```
use Benchmark;

sub one_by_one {
    my (@c);
    for (my $i; $i < 100000; $i++) {
        $c[$i] = rand;
    }
}

sub preallocate {
    my (@c);
    $#c = 99999;
    for (my $i; $i < 100000; $i++) {
        $c[$i] = rand;
    }
}

timethese (100, {    'preallocate' => 'preallocate()',
                     'one_by_one'  => 'one_by_one()'
                } );

Benchmark: timing 100 iterations of one_by_one, preallocate...
one_by_one: 111 secs (50.85 usr  0.52 sys = 51.37 cpu)
preallocate: 148 secs (67.13 usr  0.57 sys = 67.70 cpu)
```

1+1=0

```
sub preallocate {
    my (@c);
    $#c = 99999;
    for (my $i; $i < 100000; $i++) {
        $c[$i] = rand;
    }
}
```

- The answer was eventually provided by Chip Salzenberg
- Perl has a clever optimization in it
- Perl figures that `@c` got big once, so it is likely to get big again
- When `preallocate` returns, `@c` is *not* deallocated
    - The next call re-uses the same space as the last call
- And that leaves `$#c = 99999` with nothing to do
    - In fact, it's a small waste of time because it's superfluous
- So you pay the cost for your 'optimization'
    - But the gross benefit is zero because you already *had* the benefit
- One optimization plus one optimization looks like zero optimizations

1+1=0

- This bites me all the time
- For example, I'll add a file cache to a program and discover it doesn't work
  - Because the OS already has a file cache behind the scenes
- I considered going to a lot of trouble to get `Tie::File` to always write whole disk blocks
  - But there's no point, because the stdio library already does that

File Editing

```
Subject: How to edit a file most efficiently?
Date: 1998/04/27
Message-ID: <3544E019.A1F7A6D6@shell.com>

If I want to edit a file (say, remove all comment lines), I
can do this:

open IN, "myin.dat" or die: $!;
open OUT, ">myout.dat" or die: $!;
while (<IN>)
 { print OUT $_ unless (/^#/);
 }
close OUT;
close IN;
rename "myout.dat", "myin.dat";
```

`But` **`this opens two files and does a rename.`** `I suspect`
`this` **`won’t be very efficient. Is there a better way?`**
`Thanks for any advice.`

- We’ll use `Devel::SmallProf` here

Devel::SmallProf

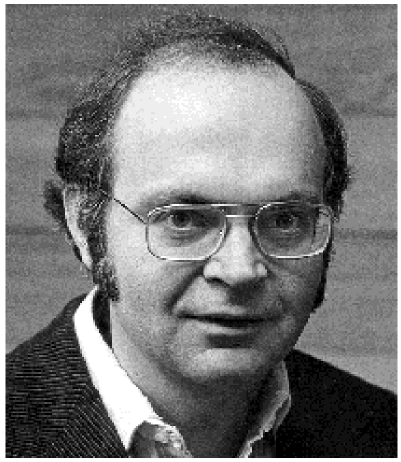
```
% wc myin.dat
1568    5707   44808 myin.dat
% perl -d:SmallProf copy1.pl
% wc myin.dat
1466    5215   40357 myin.dat
% cat smallprof.out
       ================ SmallProf version 0.9 ================
                        Profile of ./copy1.pl                    Page 1
  =================================================================
count wall tm  cpu time line
    0 0.000000 0.000000     1:#!/usr/bin/perl
    0 0.000000 0.000000     2:
    1 0.000186 0.000000     3:open IN, "myin.dat" or "die: $!";
    1 0.000196 0.000000     4:open OUT, ">myout.dat" or "die: $!";
 1570 0.013959 0.270000     5:while (<IN>)
 1569 0.015762 0.270000     6: { print OUT $_ unless (/^#/);
    0 0.000000 0.000000     7: }
    1 0.000239 0.000000     8:close OUT;
    1 0.000054 0.000000     9:close IN;
    1 0.000304 0.000000    10:rename "myout.dat", "myin.dat";
```

- Lines 5 and 6, which copy the file, consume 96.8% of the total run time
  - And so close to 100% of the CPU time that the difference is not detectable

But this opens two files and does a rename. I suspect this won’t be very efficient.
Is there a better way? Thanks for any advice.

- My advice: **`You are worrying about the wrong thing`**

Good Advice



- Donald E. Knuth, a famous wizard, is fond of saying:

  Premature optimization is the root of all evil.

- (He’s actually quoting Tony Hoare here)

Premature Optimization

- I spent a lot of time and effort writing a really good cache algorithm for `Tie::File`
- It is very sophisticated
- It uses a heap data structure to implement a least-recently-used queue
- Old records are expired from the cache when it becomes full
- A very nice piece of programming
- Unfortunately, it makes `Tie::File` slower, not faster
- At least I got my pony

Premature Optimization

- My reasoning was that `Tie::File` usage will be heavily I/O bound
- So anything I could do to reduce real I/O would speed up the module
- Having made that decision, I invested a lot of effort in a sophisticated caching algorithm
- But I was wrong
- The typical cache hit rate for programs using `Tie::File` is close to 0
- The expense of maintaining the cache is wasted
- See Bonus Slides for a quantitative analysis of caching

Vanity, Vanity, all is Vanity

- Some months ago, I asked the Philadelphia Perl Mongers

Why do people bother to use the Schwartzian Transform?

```
# Schwartzian Transform
@sorted = map { $_->[0] }
          sort { $b->[1] <=> $a->[1] }
          map { [ $_, -M $_ ] } @files;
```

- My idea was that this alternative is much easier to understand:

```
# Alternative
{ my %date;
  $date{$_} = -M $_ for @files;
  @sorted = sort { $date{$b} <=> $date{$a} } @files;
  undef %date;
}
```

- I did some benchmarks and found that it was only fractionally slower

```
NULL:   0.00u   0.00s   0.00total
ST:     8.73u   1.48s  10.21total
Hash:   9.59u   1.63s  11.22total
```

Vanity, Vanity, all is Vanity

- There was a followup:

> I decided to apply Benchmark to these various approaches. I first compiled a list of 9952 filenames, then sorted them 10**7 times ...

- Here's the code he showed:

```
timethese(10**7, {
        'CODE A' => '@sorted = sort { -M $b <=> -M $a } @filen
        'CODE B' => '@sorted = map { $_->[0] }
                               sort {$b->[1] <=> $a->[1]}
                               map {[$_, -M $_]} @filenames;',
        'CODE C' => '$date{$_} = -M $_ for @filenames;
                     @sorted = sort {$date{$b} <=> $date{$a} }
                     undef %date;',
        'CODE D' => '@sorted = map  $_->[0],
                               sort {$b->[1] <=> $a->[1]}
                               map [$_, -M $_], @filenames;',
        }
);
```

- The warning sign is already visible, although I didn't pick up on it yet

Vanity, Vanity, all is Vanity

```
Results:
Benchmark: timing 10000000 iterations of CODE A, CODE B, CODE
D...
    CODE A: 39 wallclock secs
        (38.50 usr +  0.00 sys = 38.50 CPU) @ 259740.26/s (n=1
    CODE B: 42 wallclock secs
        (42.57 usr +  0.00 sys = 42.57 CPU) @ 234907.21/s (n=1
    CODE C: 93 wallclock secs
        (91.94 usr +  0.00 sys = 91.94 CPU) @ 108766.59/s (n=1
    CODE D: 43 wallclock secs
        (42.13 usr +  0.00 sys = 42.13 CPU) @ 237360.55/s (n=1
```

- Does anyone see anything strange here?
- (The 0.00 system time is not an anomaly)
    - (This benchmark was run on a Windows system)

Vanity, Vanity, all is Vanity

I decided to apply Benchmark to these various approaches. I first compiled a list of 9952 filenames, then sorted them 10**7 times ...

- Here's the real tipoff that something is wrong

```
CODE A: 39 wallclock secs
    (38.50 usr +  0.00 sys = 38.50 CPU) @ 259740.26/s (n=1
```

- This says that his computer is sorting 9952 filenames 10000000 times in 39 seconds
- That means it's sorting 9952 filenames in 3.9 microseconds
- Not likely.

Vanity, Vanity, all is Vanity

- What went wrong here?
- The actual code was something like this:

```
my @filenames = glob("/tmp/*");

timethese(10**7, {
        'CODE A' => '@sorted = sort { -M $b <=> -M $a } @filen
        ...
);
```

- When you give strings to `Benchmark`, it executes them with `eval`
- It does the `eval` internally, inside of `Benchmark.pm`
- This is outside the scope of `my @filenames`
- The benchmark is using `@Benchmark::filenames`, which is empty
- You can indeed sort an empty list in 3.7 microseconds
- But the results were entirely meaningless

Vanity, Vanity, all is Vanity

- Anyone can make a technical error like this one
- But the real problem is more serious
- What *really* went wrong here?

  1. People using `Benchmark.pm` have a tendency to disengage their brains
     - The author of the benchmark took the obviously nonsensical results at face value
     - He wrote up a detailed analysis of these nonsensical results
  2. `Benchmark.pm` is complex
     - Here there was a scope problem that was obscured by the use of `Benchmark.pm`
     - The code wasn't doing what it appeared to be doing
  3. `Benchmark.pm`'s internals are obscure
     - This tends to inhibit understanding of the absolute numbers that it emits
     - You tend to compare the relative quantities only

Vanity, Vanity, all is Vanity

- Postscript: In 2005 I gave this class at OSCON
- An audience member interrupted to say he had found an obvious way to speed up `letter_histogram`
    - He had benchmarked it and found it substantially faster
- His benchmark looked something like this:

```
use Benchmark;
my $t = "some reasonably long string here";
timethese(-5, { orig => 'orig_letter_histogram($t)' ,
                mine => 'my_letter_histogram($t)' ,
              });

sub orig_letter_histogram {
  my $strdex = (length $_[0])-1;
  $letter_hist{substr($_[0],$_,1)}++     for (0..$strdex);
}
sub my_letter_histogram {
  $letter_hist{$1}++ while $_[0] =~ m/(.)//gs;
}
```

- The following week, I did it right
- His suggestion is 250% slower:

```
orig histo 11.42  0.03  11.45
while //gs 37.28  0.03  37.31
      NULL  0.04  0.00   0.04
```

Numerical Calculation

http://www.perlmonks.org/index.pl?node_id=134419

Good day, fellow monks. I've got a snippet of code that I'm hoping you can help me speed up. My code is to find the N-th root of a given number.

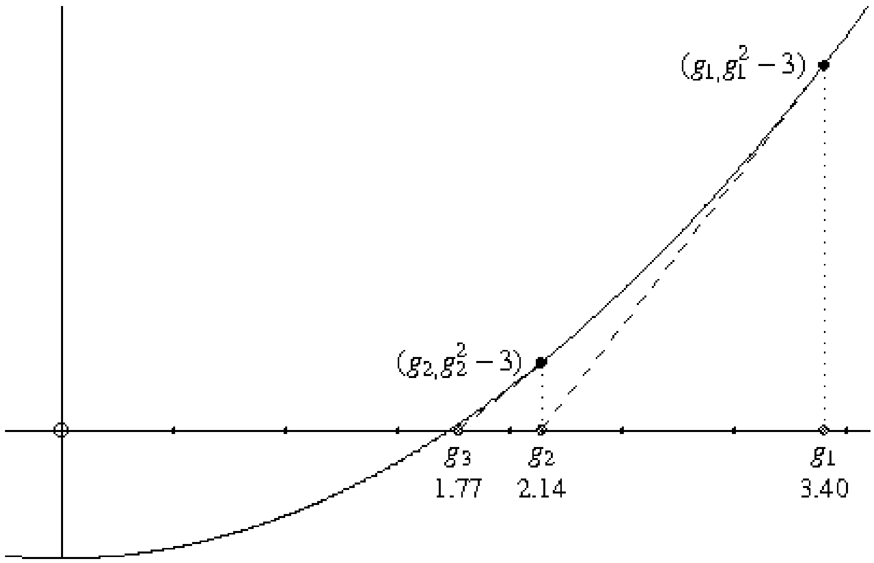
```
use Math::BigFloat;

sub Root {
    my $num        = shift;
    my $root       = shift;
    my $iterations = shift || 5;
    if ( $num < 0 ) { return undef }
    if ( $root == 0 ) { return 1 }
    my $Num = Math::BigFloat->new( $num );
    my $Root = Math::BigFloat->new( $root );
    my $current = Math::BigFloat->new();
    my $guess   = Math::BigFloat->new( $num / $root );
    my $t       = Math::BigFloat->new( $guess ** ( $root - 1 ) );
    for ( 1 .. $iterations ) {
        $current = $guess - ( $guess * $t - $Num ) / ( $Root * $t  );
        if ( $guess eq $current ) { last }
        $t = $current**($root-1);
        $guess = $current;
    }
    return $current;
}
```

This uses Newton's method for finding the roots. It produces very accurate results, provided you increase the number of iterations if you're dealing with large numbers and/or large roots. Therein lies the problem.

Numerical Calculation

- What's Newton's Method?



- Here we want to find sqrt(3)
  - This is a number x such that $x^2 - 3 = 0$
  - That's the x-coordinate of the point where the parabola crosses the x-axis
- Make a guess g_1
  - Extend the tangent to the parabola at g_1 until it intersects the axis
  - This is g_2, which is a better guess than g_1 was
  - Repeat as desired

Numerical Calculation

If you want something relatively simple like the 5th root of 100:

```
$x = Root( 100, 5 );
```

the result is reasonably fast. However, with each iteration, it get progressively slower. So if you wanted something enormous, like:

```
$x = Root( 500000, 555 );
```

you could be waiting for ages. If we leave the number of iterations low, the result will likely be very inaccurate, but as we increase the number of iterations, each individual iteration gets slower and slower. The only thing I’ve been able to come up with so far is the comparison of `$guess` and `$current` inside the for loop. I was able to get a bit of a speed boost by doing a string comparison rather than a numeric comparison. Any suggestions on how to speed this up?

Numerical Calculation

- There were a whole load of pointless suggestions:

```
BTW It seems that using Math::BigFloat methods directly is
slighly faster then relying on overloaded operations:

 timethese(1000, {
    Methods => sub { Math::BigFloat->new(100)->fmul(Math::BigFloat->new(100)) },
    Operations => sub { Math::BigFloat->new(100) * Math::BigFloat->new(100) },
});

Benchmark: timing 1000 iterations of Methods, Operations...
   Methods:  2 wallclock secs
   ( 1.48 usr +  0.01 sys =  1.49 CPU)
   @ 671.14/s (n=1000)
   Operations:  1 wallclock secs
   ( 1.63 usr +  0.00 sys =  1.63 CPU)
   @ 613.50/s (n=1000)
```

- This guy just couldn't leave well enough alone:

```
Moreover using subroutine calls should be even more
faster. That is use Math::BigFloat::OP($num) instead of
       $num->OP.
```

Numerical Calculation

```
Let me throw around my math skills... Recalling some binary
math I figured that 10^2 (10 to the power of 2) may be
simplified into this: 10^2 = 10x10 = 10x(2x2x2+2).

Notice all the 2's there? Here's where the left shift
operator '<<' comes in handy (and it's pretty fast by the
way).

So, every multiplication by 2 could be replaced by a left
shift by one (in binary it's equivalent to multiplying by 2
;) like this:

10^2 = 10<<3 + 10<<1; (by the way, this is may not be written
as 10<<4! :)

So, I've replaced 10x10 by a few left shift operators. The
key here is to determine how many left shifts will have to be
performed for given power.
```

- Etc.
- Now, if we were programming in assembly language, maybe
  - (Maybe not)

Numerical Calculation

- You should really check out this thread
    - It's a gold mine of bad advice
- One guy even threw up his hands:

```
Without delving into the internals of Math::BigFloat, I don't
see any way to speed this up. Perhaps you could try a
different approach? A different algorythm maybe?
```

- And that was probably the least worthless suggestion
- Except for (ahem) mine

Numerical Calculation

- First, what about this?

  > as we increase the number of iterations, each individual iteration gets slower and slower.

- Suppose you have two numbers of 8 decimal places each
  - Say 0.12345678 and 0.23456789
- What happens when you multiply them?
  - You get 0.0289589963907942, which has 16 digits
- If you multiply two 16-digit numbers, you get a 32-digit result
- `Math::BigFloat` never throws away any trailing digits
  - The numbers get longer and longer every time you do a multiplication

Numerical Calculation

- Newton’s method takes a guess and finds a better guess
  - The number of correct bits in the guess tends to double on each iteration
  - If the initial guess is good, the new guess is superb
  - If there were no correct bits to begin with, it wanders around aimlessly
- The initial guess in the original code was **terrible**:

```
my $guess   = Math::BigFloat->new( $num / $root );
```

- For `Root(500000, 555)` this guesses that the root is 900.9009009
  - The root is actually 1.02392563097332211627
- At x = 900.9, the curve $y = x^{555}$ - 500000 is **extremely** steep
  - The tangent line is almost vertical (it has a slope of about 3.4e1639)
  - So the ’improved’ guess is almost the same as the original guess
  - But twice as long!

Numerical Calculation

- Instead of making a lousy initial guess, like this:

```
my $guess   = Math::BigFloat->new( $num / $root );
```

- Make a good initial guess, like this:

```
my $guess   = Math::BigFloat->new( $num ** (1/$root) );
```

- This uses the hardware floating-point arithmetic to calculate the right answer...
  - ...to 53 bits of accuracy...
  - ...instantaneously
- Then use Newton's method to get even closer
- After 4 iterations, you have 130 decimal places correct
- Moral of the story: Stop fussing around with micro-optimizations
- Second moral: The world is full of crappy optimization advice

Crappy Advice

- The following appeared on the StLouis.pm web page last year:

```
Perl Tip: Use each when iterating through a hash
table. It's far better than keys for iterating over large
hash tables.
```

- Better for what? Curing sciatica?
- Supposing the author meant 'faster', he was wrong

`each` vs. `keys`

- This gets the keys all at once, in C:

```
for (keys %hash) {
  ...
}
```

- This gets the keys one at a time, dispatching Perl operations in between:

```
while (my $k = each %hash) {
  ...
}
```

- The purpose of `each` is to conserve *space*, not time
- You use it when the hash is very large and you don't want to store all the keys at once
    - For example if the hash is tied to a large disk file
- Since it is a space-conserving optimization, you would expect it to be slower than `keys`
- And so it is
    - Unless you're also interested in the values
    - Or unless the `keys` call causes your program to become memory-bound

What to Remember

(Antepenultimate slide)

1. Look at the big picture first - think about the project, not the program
2. It's hard to guess what part of the program matters, so use tools
3. 90% of the runtime is accounted for by 10% of the code
4. The speed of the other 90% of the code hardly matters at all...
   - ...so don't waste your time on it
5. The `Benchmark` module is good for answering questions that aren't worth asking

Jackson's Rules

- All this was summed up by famous computer scientist Michael A. Jackson
- In his "Two rules of when to optimize"
    - (*Principles of Program Design*, 1975)

Jackson’s Rules

1. Don’t do it.

Jackson's Rules

2. (For experts only)

Don't do it yet.

Thank You

- Questions? Send me mail.

  `mjd-tpc-perf+@plover.com`

Bonus Slides

- Writing a class is like making a film
- Some good stuff ends up on the floor of the editing room
- If this class were a DVD, this stuff would be the "special features and deleted scenes"

Pod::ParseTree::append

- Results:
  - Before: 4th place

```
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 5.90   1.179  1.134   7733   0.0002 0.0001  Pod::ParseTree::append
```

- After: 10th place

```
2.22   0.370  0.327   7733   0.0000 0.0000  Pod::ParseTree::append
```

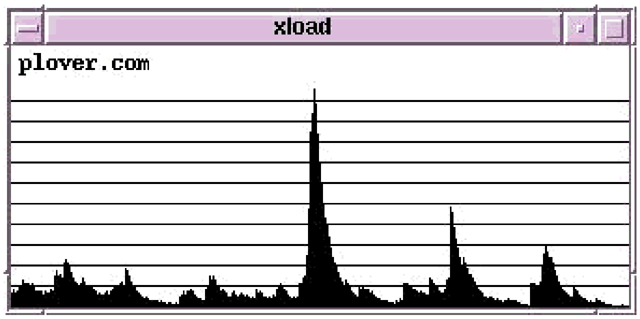
- Note that it’s 2.22% of the new *shorter* run time
- The new `append` would have been in 16th place in the ’before’ version
- End of digression

System Load

- System administrators are interested in *system load*
- This is what is reported by the `uptime` command:

```
7:19pm  up 65 days,  7:23, 24 users,  load average: 0.22, 0.44, 0.81
```

- And by tools like `xload`



- It is the average number of jobs that are ready to be run
  - (This omits jobs that are sleeping, waiting for I/O, etc.)
- If it exceeds the number of CPUs, then the system is overloaded

Memory Bound Programs

- Here are the values I plotted in the graphs:

Input Size	Wallclock time
1000	0.10
2000	0.14
4000	0.30
8000	0.62
16000	1.43
32000	3.05
64000	6.19
128000	12.50
256000	28.19
512000	71.69
1024000	134.87
2048000	14601.00

Memory Bound Programs

- Here's the raw data for the last three lines
  - I made three runs with each size and took the median run time

512 000

```
24.91user 0.85system 1:11.69elapsed 35%CPU
  (256major+7597minor)pagefaults 0swaps
25.09user 0.56system 0:54.91elapsed 46%CPU
  (256major+7597minor)pagefaults 0swaps
27.62user 0.64system 1:13.82elapsed 38%CPU
  (256major+7597minor)pagefaults 0swaps
```

1 024 000

```
60.38user 1.90system 2:06.18elapsed 49%CPU
  (299major+19082minor)pagefaults 0swaps
71.49user 1.80system 2:14.87elapsed 54%CPU
  (256major+15156minor)pagefaults 0swaps
74.08user 1.56system 2:21.39elapsed 53%CPU
  (256major+15156minor)pagefaults 0swaps
```

2 048 000

```
251.00user 120.34system 4:38:10elapsed 2%CPU
  (487major+1065900minor)pagefaults 0swaps
214.70user 86.19system 3:01:45elapsed 2%CPU
  (486major+803641minor)pagefaults 0swaps
256.03user 98.89system 4:03:21elapsed 2%CPU
  (486major+880664minor)pagefaults 0swaps
```

- Notice how the user time increases moderately and the system time explodes

What’s Memoization?

- Memoization replaces a function `f` with a *stub*, `m`
- `m` manages a *cache*
- If the desired value of `f` is in the cache, it is returned
  - (*Cache hit*)
- If not, `f` is called and the value is stored in the cache
  - (*Cache miss*)
- It is a speed optimization - trades space for time

Walt's Dilemma

- My friend Walt wrote a program to solve a math puzzle
  - Find 'excellent numbers' like `190476` or `48`

$476^2 - 190^2 = 226576 - 36100 = 190476$

$8^2 - 4^2 = 64 - 16 = 48$

- Walt's program had

```
sub square { return $_[0] * $_[0] }
```

- Since `square` was called a lot, he memoized it
- Now the program was slower
- Here's why

How Long Does It Take?

- Question: Will the memoized function be faster than the original?
- It depends on:
  - How long the original function `f` takes
  - How often `f` is actually called
  - How long the cache management takes

Cache Hit Rate

- Suppose we make some calls to `m`, the stub
- We find that 37% of the time, the desired value is already in the cache
- The other 63% of the time, the real `f` must be called
- We have a *cache hit rate* of 0.37
- Hit rate is always between 0 and 1
  - 1: A cached value is available every time; `f` is never called
  - 0: The cached value is never there

Time to Call a Memoized Function

- Let's suppose we make N calls to `m`
- Suppose the cache hit rate is h
  - Cache *miss* rate is $1-h$
  - The real `f` gets called about $N(1-h)$ times
- Suppose the average time for `f` to execute is f
  - Time spent in `f` is $N(1-h)f$
- Suppose the average time to manage the cache is K
  - Time spent managing the cache is NK
- Total time spent for N calls: $N(1-h)f + NK$
  - Average time per call: $(1-h)f + K$

Time Savings

- h is the hit rate
- f is the time it takes to call the original function
- K is the average cache management overhead
- Average time spent per call to `m`: $(1-h)f + K$
- The average time for the *un*memoized function is f
- Time saved (per call) by memoizing: $f - (1-h)f - K$
  - Equals $hf - K$
- hf is the *benefit*. K is the *cost*.
- We want $hf > K$

For Example...

Time saved is *hf* - *K*

- High cache hit rate *h* leads to larger savings
- Large function call overhead *f* leads to larger savings
- Large cache management overhead *K* leads to smaller savings
- Typically, *h* and *f* are not under anyone's control
- The best strategy for the author of `Memoize` is to make *K* as small as possible

For Example...

We win if $hf > K$

- Suppose hit rate h is 0 ?

For Example...

We win if $hf > K$

- Suppose K is bigger than f
- Buf hf is *smaller* than f
- We lose!
- We can tolerate large cache management overhead...
  - But only if the function takes a really long time
  - If f is real big, it's easier to get a win
  - In spite of a big K

For Example...

We win if $hf > K$

- Suppose f is really really small
- hf is even smaller
  - Perhaps close to zero
- We can't win in such a case
- As Walt unfortunately found out
- In Walt's program, f was the time to do one multiplication
  - This is a budget of time that K must not exceed
  - If K does even one multiplication, it blows the budget

Devel::DProf

- Here's the contents of `tmon.out`
- First there's a header section with metainformation:

```
#fOrTyTwO
$hz=100;
$XS_VERSION='DProf 20000000.00_01';
# All values are given in HZ
$over_utime=11; $over_stime=1; $over_rtime=12;
$over_tests=10000;
$rrun_utime=746; $rrun_stime=7; $rrun_rtime=800;
$total_marks=33941
```

- `$hz` is the clock resolution of the system
  - Here one 'Hz' is 1/100 second
- The `$over_` variables try to record overhead of checking the clock
  - (`u` == user time, `s` == system time, `r` == real (wallclock) time)
  - For example, 11/10000 user-seconds per call
- `$rrun_` are the total times consumed by the sample run
- `$total_marks` is the total number of subroutine entries and exits

Devel::DProf

```
& 1b Mail::Header fold_length
+ 1b
- 1b
...
& 21 Mail::Header _fmt_line
+ 21
& 22 Mail::Header _tag_case
+ 22
- 22
+ 1b
- 1b
& 23 Mail::Header _fold_line
+ 23
@ 0 0 4
- 23
- 21
& 24 Mail::Header _insert
+ 24
- 24
+ 21
+ 22
- 22
+ 1b
- 1b
```

- `&` lines assign a new ID number to a subroutine
- `+` and `-` indicate that the subroutine was entered or exited
- `@` lines indicate that the indicated number of ticks elapsed since the last `@` line

parse_text

```
if ( /^([A-Z])<([^<>]*)>$/ ) {
    $seq = Pod::InteriorSequence->new(
               -name   => $1,
               -ldelim => "<",  -rdelim => ">",
               -file   => $file,    -line   => $line
           );
    $seq->append($2);
    $seq->nested($seq_stack[-1]) if @seq_stack > 1;
    $seq_stack[-1]->append($expand_seq
                           ? &$xseq_sub($self, $seq)
                           : $seq);
}
```

- What was this about?
- We’re building a tree of `Pod::InteriorSequence` nodes
    - In `X<Y<...>>`, node `Y` is a child of node `X`
    - The `->nested` call installs a pointer to `X` into `Y`

Slide Manufacturing

- `make-slides` takes a single file with slides
  - Slides are separated by rows of hyphens
- Slides are written out to a series of separate text files
  - `text2slide` is run on each of these files
- There are some other features as well
- Let's see what we can do with it
- Unfortunately it has few subroutines, so `Devel::DProf` isn't much help

```
Total Elapsed Time = 60.50918 Seconds
  User+System Time = 0.799279 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 6.26   0.050  0.050      2   0.0250 0.0248  main::BEGIN
 0.00   0.000 -0.000      2   0.0000      -  Exporter::import
 0.00   0.000 -0.000      2   0.0000      -  File::Glob::BEGIN
 0.00   0.000 -0.000      1   0.0000      -  strict::import
 0.00   0.000 -0.000      1   0.0000      -  strict::bits
 ...
```

Slide Manufacturing

- `SmallProf` does produce some useful results, however
- Here's the data sorted by CPU time:

```
count wall tm  cpu time line
   83 73.67689 31.70000   287:  system $cmd;
 1960 0.637309 0.340000    69:  if (/^(\t|\s{5})/ && !($DIVERSION{active}
 1896 0.659382 0.280000    94:  if (/^-{12}/ || /^={32}/) {
 1960 0.307705 0.240000    68:  s{%(\w+)%}{exists $macro{$1} ? $macro{$1} :
 1813 0.330660 0.210000   210:  $accumulated .= $_ unless $skip_this;
 1960 0.275886 0.200000    83:  if ($MACROS && s/^\#MACRO\#\s+//) {
 1960 0.337091 0.180000    67:while (<STDIN>) {
 1960 0.250034 0.180000    91:  next if /^\#{3}/;
 1813 0.290792 0.180000   202:  if ($DIVERSION{active} && (/^(\#\*|  )   /
 1897 0.284703 0.170000    92:  last if /^-{50,}END/;
  524 0.464508 0.150000    74:    $length -= 3 while / \[\w\[ | \]\w\]/xg;
  524 0.233224 0.100000    73:    $length -= 2 while / \[  \[ | \]  \]/xg;
```

- Clearly, run time is dominated by line 287

```
285    my $cmd = qq{$TXT2HTML $enc
                    --setvar MJD_FIRST_FILE=$firsthtml
                    --setvar MJD_LAST_FILE=$lasthtml
                    --setvar MJD_NEXT_FILE=$nexthtml
                    --setvar MJD_PREV_FILE=$prevhtml
                    --setvar MJD_SLIDE_NUMBER=$slideno
                    --title '$title' $slide > $html};
286  #  print STDERR "Command: $cmd\n";
287    system $cmd;
```

- There's probably not too much we can do about this

Mail Folder Analyzer Revisited

- Now that we've sped up the analyzer by a factor of 6, let's see what else we can do
- We'll rerun the test under the profiler

```
 % perl -d:DProf mfa2.pl MBOX  > /dev/null
 % dprofpp
```

```
Total Elapsed Time = 1.492102 Seconds
  User+System Time = 1.302102 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 63.7   0.830  0.829    109   0.0076 0.0076  main::letter_histogram
 13.8   0.180  0.180      1   0.1800 0.1800  Mail::Util::read_mbox
 4.61   0.060  0.130      3   0.0200 0.0432  main::BEGIN
 4.61   0.060  0.887    109   0.0005 0.0081  main::handle_message
 2.30   0.030  0.030      1   0.0300 0.0300  warnings::BEGIN
 2.30   0.030  0.040      5   0.0060 0.0080  Mail::Util::BEGIN
 0.77   0.010  0.010      3   0.0033 0.0033  AutoLoader::BEGIN
 0.77   0.010  0.010      2   0.0050 0.0050  main::pairify
 0.77   0.010  0.010      4   0.0025 0.0025  vars::BEGIN
 0.00   0.000 -0.000      3   0.0000      -  strict::import
 0.00   0.000 -0.000      3   0.0000      -  strict::bits
 0.00   0.000 -0.000      2   0.0000      -  Exporter::import
 0.00   0.000 -0.000      1   0.0000      -  warnings::import
 0.00   0.000 -0.000      1   0.0000      -  warnings::register::import
 0.00   0.000 -0.000      2   0.0000      -  warnings::register::mkMask
```

- We see that `letter_histogram` is run 109 times at 7.6 ms each
    - This is 64% of the remaining run time

The Innermost Loop

```
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 63.7   0.830  0.829    109   0.0076 0.0076  main::letter_histogram
```

- This is a typical situation
    - Often, a program is structured as a series of nested loops
- For example, this program:
    - For each input file,
        - For each message in the file,
            - For each character in the file
                - Append it to the histogram.
- The code inside the innermost loop gets run many, many times
    - Here, once for each character in the entire input
- Other parts of the program are run much less frequently
    - A small speedup in this innermost loop can have a disproportionate effect on run time

The 64% Question

```
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 63.7   0.830  0.829    109   0.0076 0.0076  main::letter_histogram
```

- I said "This is 64% of the remaining run time"
  - Why 64% and not 63.7% ?
- The total run time was about 1.22 CPU seconds
  - The resolution of the measurements was only 0.01 second
  - The resolution of the `%Time` column is therefore 0.81%
- It’s like announcing that "85.714% of surveyed respondents prefer Perl to Python"
  - Sounds really precise
  - But what you actually mean is "6 out of 7"
- That `63.7%` actually means "83 out of 122"
  - Or perhaps "somewhere between 62.9 and 64.5%"
- The percentages are reported with eight times more precision than the measurements actually have

The 64% Question

- Scientists and engineers are trained to deal with this
  - They know that 3 meters is different from 3.000 meters
  - One was measured to a precision of 1 meter, the other to a precision of 1 mm
- They get training in how to calculate with imprecise measurements
  - How to represent and understand the error ranges
  - How to present the answers without lying
- Computer programmers are not usually so trained
- I would like to see CS curricula revised to fix this
- I would like to see computer 'science' as a real science

The 90-10 Rule In Action

- Counting modules, the program has 2,848 lines of code
  - (I didn't count whitespace, comments, POD, lines with just braces, etc.)

Subroutine	%time	cum.	lines	cum.	cum%
M::H::_fold_line	30.1	30.1	48	48	1.7
M::H::_fmt_line	24.6	54.7	34	82	2.9
letter_histogram	12.2	66.9	3	85	3.0
M::H::_insert	6.6	73.5	22	107	3.8
M::H::extract	5.8	79.3	16	123	4.3
M::H::_tag_case	5.3	82.6	6	129	4.5
M::H::fold_length	5.3	87.9	15	144	5.1
M::H::fold	3.7	91.6	13	157	5.5
M::U::read_mbox	2.4	94.0	21	178	6.3
M::I::BEGIN	1.4	95.4	18	196	6.9
...					

- 7% of the code accounts for more than 95% of the run time
  - 5% of the code accounts for more than 80% of the run time

The 90-10 Rule In Action

- Perhaps counting modules biased the numbers?

```
Subroutine          time %time  cum.    lines  cum.  (%)
letter_histogram    .76  79.2   79.2        3     3     5.9
BEGIN               .10  10.4   89.6       13    16    31.4
handle_message      .09   9.4   99.0        7    23    45.1
report              .01   1.0  100.0       20    43    84.3
from_histogram      .00   0.0  100.0        3    46    90.2
pairify             .00   0.0  100.0        5    51   100.0
```

- No, we still have 6% of the code accounting for 80% of the run time

Error Variation

- I ran five identical runs of the same program on the same input:

```
User+System Time = 1.252102 Seconds
User+System Time = 1.260666 Seconds
User+System Time = 1.280666 Seconds
User+System Time = 1.319948 Seconds
User+System Time = 1.309948 Seconds
```

- That’s more than 5% variation

```
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 65.4   0.820  0.819    109   0.0075 0.0075  main::letter_histogram
 63.4   0.800  0.799    109   0.0073 0.0073  main::letter_histogram
 62.4   0.800  0.799    109   0.0073 0.0073  main::letter_histogram
 65.1   0.860  0.859    109   0.0079 0.0079  main::letter_histogram
 64.8   0.850  0.849    109   0.0078 0.0078  main::letter_histogram
```

- Ditto
- Conclusion: Don’t put any faith in the exact numbers
- Corollary: If someone tells you that *X* is 5% faster than *Y*, ignore them

Error Variation

```
Date: Tue, 1 Jan 2002 14:46:06 +0100
Subject: Re: How can I determine a 0 byte File
Message-Id: <a0seef$668$05$1@news.t-online.com>

timethese( $count, {
   'stat' => sub { (stat($filename))[7] },
   'z'    => sub {  -z $filename },
   's'    => sub {  -s $filename },
} );

Benchmark: timing 100000 iterations of s, stat, z...
s:    48 wallclock secs (11.49 usr + 29.25 sys = 40.74 CPU)
      @ 2454.65/s (n=100000)
stat: 53 wallclock secs (14.21 usr + 30.65 sys = 44.87 CPU)
      @ 2228.91/s (n=100000)
z:    50 wallclock secs (11.66 usr + 29.76 sys = 41.42 CPU)
      @ 2414.35/s (n=100000)

Stat indeed seems to be a little slower...
```

- I think that's the wrong conclusion

```
...but then, if -s is faster than -z, the whole difference
may be within the error margin.
```

- I think that's the right conclusion

Mail Folder Analyzer Revisited

- Back to the MFA
- The profiler says that `main::letter_histogram` is consuming most of the CPU time

```
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 63.7   0.830  0.829    109   0.0076 0.0076  main::letter_histogram
 13.8   0.180  0.180      1   0.1800 0.1800  Mail::Util::read_mbox
```

- A 20% speedup in this one function would reduce the program's run time by 1/8

```
sub letter_histogram {
  my $strdex = (length $_[0])-1;
  $letter_hist{substr($_[0],$_,1)}++     for (0..$strdex);
}
```

- Perhaps loop over the characters directly
  - Instead of looping over `0 .. $strdex` and indexing the string?

```
sub letter_histogram {
  $letter_hist{$_}++     for split //, $_[0];
}

Before                     After

real    0m2.739s           real    0m5.379s
user    0m1.270s           user    0m2.410s
sys     0m0.040s           sys     0m0.040s
```

- Well, that didn't work

letter_histogram

```
sub letter_histogram {
  my $strdex = (length $_[0])-1;
  $letter_hist{substr($_[0],$_,1)}++     for (0..$strdex);
}
```

- Perhaps we could get a speedup by avoiding the repeated array lookup on `@_`?

```
sub letter_histogram {
  my $msg = shift;
  my $strdex = (length $msg)-1;
  $letter_hist{substr($msg,$_,1)}++     for (0..$strdex);
}
```

- Cost: `shift` plus an extra copy of the data

```
Before                         After

real    0m1.277s               real    0m1.236s
user    0m1.250s               user    0m1.220s
sys     0m0.020s               sys     0m0.010s
```

- No significant difference
  - Perhaps it really is .04 ms faster
  - But who the heck cares?
- Other things I tried:
  - Use `@letter_hist` instead of `%letter_hist`
  - Call `letter_histogram` once on entire mbox instead of on each message

Good Advice

- Actually in 1998 I had a little more to say:

> Worrying about optimization at this level is just silly. Write the program. If it is unacceptably slow for your real application, then benchmark it, and then look at ways to make the slow parts faster.

- I think this is the best general advice you can get about optimization
  - Hence this class

Good Advice

- Here's some advice that is more Perl-specific

If you're worried about the slowness of two `opens` and a `rename`, why aren't you worried about the much greater slowness of `perl`?

- It's important to keep these things in perspective
    - If you're really worried about the cost of a single `rename`, you are using the wrong language

```
int main(void) {
  int i, j;
  long total;
  for (i=0; i<1000; i++) {
    total = 0;
    for (j=0; j<1000; j++) {
      total += j;
    }
  }
  printf("%ld\n", total);
}

real    0m0.071s
user    0m0.060s
sys     0m0.000s
```

```
my $total;
for (0 .. 999) {
  $total = 0;
  for my $j (0 .. 999) {
    $total += $j;
  }
}
print $total, "\n";

real    0m2.493s
user    0m2.340s
sys     0m0.020s
```

- The C version was **35** times faster

Good Advice

- Donald E. Knuth, a famous wizard, is fond of saying:

Premature optimization is the root of all evil.

- Here's some context:

There is no doubt that the "grail" of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and such attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, about 97% of the time. Premature optimization is the root of all evil.

- He continues:

Yet we should not pass up out opportunities in that critical 3%. Good programmers will not be lulled into complacency by such reasoning, they will be wise to look carefully at the critical code; but only *after* the critical code has been identified.