Devel::NYTProf

Perl Source Code Profiler

Tim Bunce - July 2010
Devel::DProf Is Broken

```perl
$ perl -we 'print "sub s$_ { sqrt(42) for 1..100 }; 
s$_({});\n" for 1..1000' > x.pl

$ perl -d:DProf x.pl

$ dprofpp -r
Total Elapsed Time = 0.108 Seconds
Real Time = 0.108 Seconds

Exclusive Times

<table>
<thead>
<tr>
<th>%Time</th>
<th>ExclSec</th>
<th>CumulS</th>
<th>#Calls</th>
<th>sec/call</th>
<th>Csec/c</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.26</td>
<td>0.010</td>
<td>0.010</td>
<td>1</td>
<td>0.0100</td>
<td>0.0100</td>
<td>main::s76</td>
</tr>
<tr>
<td>9.26</td>
<td>0.010</td>
<td>0.010</td>
<td>1</td>
<td>0.0100</td>
<td>0.0100</td>
<td>main::s323</td>
</tr>
<tr>
<td>9.26</td>
<td>0.010</td>
<td>0.010</td>
<td>1</td>
<td>0.0100</td>
<td>0.0100</td>
<td>main::s626</td>
</tr>
<tr>
<td>9.26</td>
<td>0.010</td>
<td>0.010</td>
<td>1</td>
<td>0.0100</td>
<td>0.0100</td>
<td>main::s936</td>
</tr>
<tr>
<td>0.00</td>
<td>- -0.000</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>main::s77</td>
</tr>
<tr>
<td>0.00</td>
<td>- -0.000</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>main::s82</td>
</tr>
</tbody>
</table>
```
Evolution

<table>
<thead>
<tr>
<th>Module</th>
<th>Year</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Devel::DProf</td>
<td>1995</td>
<td>Subroutine</td>
</tr>
<tr>
<td><strong>Devel::SmallProf</strong></td>
<td>1997</td>
<td>Line</td>
</tr>
<tr>
<td>Devel::AutoProfiler</td>
<td>2002</td>
<td>Subroutine</td>
</tr>
<tr>
<td>Devel::Profiler</td>
<td>2002</td>
<td>Subroutine</td>
</tr>
<tr>
<td>Devel::Profile</td>
<td>2003</td>
<td>Subroutine</td>
</tr>
<tr>
<td><strong>Devel::FastProf</strong></td>
<td>2005</td>
<td>Line</td>
</tr>
<tr>
<td>Devel::DProfLB</td>
<td>2006</td>
<td>Subroutine</td>
</tr>
<tr>
<td>Devel::WxProf</td>
<td>2008</td>
<td>Subroutine</td>
</tr>
<tr>
<td>Devel::Profilt</td>
<td>2008</td>
<td>Line</td>
</tr>
<tr>
<td><strong>Devel::NYTProf v1</strong></td>
<td>2008</td>
<td>Line</td>
</tr>
<tr>
<td><strong>Devel::NYTProf v2</strong></td>
<td>2008</td>
<td>Line &amp; Subroutine</td>
</tr>
<tr>
<td><strong>Devel::NYTProf v3</strong></td>
<td>2009</td>
<td>Line &amp; Sub &amp; Opcode</td>
</tr>
<tr>
<td><strong>Devel::NYTProf v4</strong></td>
<td>2010</td>
<td>Line &amp; Sub &amp; Opcode</td>
</tr>
</tbody>
</table>
Profiling 101

The Basics
What To Measure?

<table>
<thead>
<tr>
<th></th>
<th>CPU Time</th>
<th>Real Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subroutines</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Statements</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
CPU Time vs Real Time

• CPU time
  - Measures time CPU sent executing your code
  - Not (much) affected by other load on system
  - Doesn’t include time spent waiting for i/o etc.

• Real time
  - Measures the elapsed time-of-day
  - Your time is affected by other load on system
  - Includes time spent waiting for i/o etc.
Subroutine vs Statement

• Subroutine Profiling
  – Measures time between subroutine entry and exit
  – That’s the Inclusive time. Exclusive by subtraction.
  – Reasonably fast, reasonably small data files

• Problems
  – Can be confused by funky control flow (goto &sub)
  – No insight into where time spent within large subs
  – Doesn’t measure code outside of a sub
Subroutine vs Statement

• Line/Statement profiling
  - Measure time from start of one statement to next
  - Exclusive time (except includes built-ins & xsubs)
  - Fine grained detail

• Problems
  - Very expensive in CPU & I/O
  - Assigns too much time to some statements
  - Too much detail for large subs
  - Hard to get overall subroutine times
Devel::NYTProf
v1 Innovations

• Fork by Adam Kaplan of Devel::FastProf
  – working at the New York Times
• HTML report borrowed from Devel::Cover
• More accurate: Discounts profiler overhead including cost of writing to the file
• Test suite!
v2 Innovations

• Profiles time *per block*!
  – Statement times can be aggregated to *enclosing block* and *enclosing sub*

• Dual Profilers!
  – Is a statement profiler *and* a subroutine profiler concurrently
v2 Innovations

• Subroutine profiler
  – tracks subroutine time *per calling location*
  – even for xsubs
  – calculates exclusive time on-the-fly
  – discounts cost of statement profiler
  – immune from funky control flow
  – in memory, writes to file at end
  – extremely fast
v2 Innovations

• Statement profiler gives correct timing after leave ops
  – last statement in loops doesn’t accumulate time spent evaluating the condition
  – last statement in subs doesn’t accumulate time spent in remainder of calling statement
  – previous statement profilers didn’t fix this
v2 Other Features

- Profiles compile-time activity
- Profiling can be enabled & disabled on the fly
- Handles forks with no overhead
- Correct timing for mod_perl
- Sub-microsecond resolution
- Multiple clocks, including high-res CPU time
- Can snapshot source code & evals into profile
- Built-in zip compression
## Profiling Performance

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perl</td>
<td>x 1</td>
<td>-</td>
</tr>
<tr>
<td>DProf</td>
<td>x 4.9</td>
<td>60,736KB</td>
</tr>
<tr>
<td>SmallProf</td>
<td>x 22.0</td>
<td>-</td>
</tr>
<tr>
<td>FastProf</td>
<td>x 6.3</td>
<td>42,927KB</td>
</tr>
<tr>
<td>NYTProf</td>
<td>x 3.9</td>
<td>11,174KB</td>
</tr>
<tr>
<td>+ blocks=0</td>
<td>x 3.5</td>
<td>9,628KB</td>
</tr>
<tr>
<td>+ stmts=0</td>
<td>x 2.5</td>
<td>205KB</td>
</tr>
</tbody>
</table>

NYTProf v2.0 running perl 5.6.8 perlcritic 1.088 on lib/Perl/Critic/Policy
v3 Features

- Profiles slow opcodes: system calls, regexps, ...
- Subroutine caller name noted, for call-graph
- Handles `goto &sub;` e.g. AUTOLOAD
- HTML report includes interactive TreeMaps
- Outputs call-graph in Graphviz dot format
- High resolution timer on Mac OS X
- Merge multiple profiles
v4 Features

• Profile reporting of code inside string evals
• Smart handling of high numbers of evals
• Smart handling of duplicate anon subs
• Better handling of assorted edge-cases
• Detection of slow regex match vars: $& $' $`
Running NYTProf

perl -d:NYTProf ...

perl -MDevel::NYTProf ...

PERL5OPT=-d:NYTProf

NYTPROF=file=/tmp/nytprof.out:addpid=1:slowops=1
Reporting: KCachegrind

- KCachegrind call graph - new and cool
  - contributed by C. L. Kao.
  - requires KCachegrind

$ nytprofcg  # generates nytprof.callgraph
$ kcachegrind # load the file via the gui
Reporting: HTML

- HTML report
  - page per source file, annotated with times and links
  - subroutine index table with sortable columns
  - interactive Treemaps of subroutine times
  - generates Graphviz dot file of call graph

$ nytprofhtml # writes HTML report in ./nytprof/...
$ nytprofhtml --file=/tmp/nytprof.out.793 --open
Profile of /usr/local/perl510-pure/bin/perldoc for 3.18s, executing 3629 statements and 1655 subroutine calls in 34 source files and 8 string evals.

Top 15 Subroutines — ordered by exclusive time

<table>
<thead>
<tr>
<th>Calls</th>
<th>P</th>
<th>F</th>
<th>Exclusive Time</th>
<th>Inclusive Time</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.87s</td>
<td>1.87s</td>
<td>Pod::Perldoc::ToMan::parse_from_file</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>637ms</td>
<td>750ms</td>
<td>Pod::Perldoc::new_tempfile</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>108ms</td>
<td>169ms</td>
<td>Pod::Perldoc::find_good_formatter_class</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>82.9ms</td>
<td>83.6ms</td>
<td>Pod::Perldoc::opt_o_with</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>76.6ms</td>
<td>77.0ms</td>
<td>Pod::Perldoc::maybe_diddle_INC</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>60.3s</td>
<td>110ms</td>
<td>base::import</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>54.5ms</td>
<td>54.5ms</td>
<td>XSLoader::load</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>52.1ms</td>
<td>52.1ms</td>
<td>Pod::Perldoc::CORE::sleep(xsub)</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>49.4ms</td>
<td>49.4ms</td>
<td>base::has_version</td>
</tr>
<tr>
<td>202</td>
<td>4</td>
<td>2</td>
<td>22.5ms</td>
<td>22.5ms</td>
<td>Pod::Perldoc::CORE::readline(xsub)</td>
</tr>
<tr>
<td>189</td>
<td>1</td>
<td>2</td>
<td>4.17ms</td>
<td>4.17ms</td>
<td>Pod::Perldoc::CORE::print(xsub)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2.81ms</td>
<td>2.88ms</td>
<td>Exporter::as_heavy</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>9</td>
<td>2.71ms</td>
<td>3.24ms</td>
<td>Exporter::import</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.96ms</td>
<td>28.8ms</td>
<td>Pod::Perldoc::page</td>
</tr>
<tr>
<td>41</td>
<td>3</td>
<td>1</td>
<td>1.18ms</td>
<td>1.70ms</td>
<td>File::Spec::Unix::canonpath</td>
</tr>
</tbody>
</table>

See all 407 subroutines
Summary

Profile of /usr/local/perl510-pure/bin/perldoc for 3.18s, executing 3629 statements and 1655 subroutine calls in 34 source files and 8 string evals.

Links to annotated source code

Timings for perl builtins

Table: Top 15 Subroutines — ordered by exclusive time

<table>
<thead>
<tr>
<th>Calls</th>
<th>P</th>
<th>F</th>
<th>Exclusive Time</th>
<th>Inclusive Time</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.87s</td>
<td>1.87s</td>
<td>Pod::Perldoc::ToMan::parse_from_file</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>637ms</td>
<td>750ms</td>
<td>Pod::Perldoc::new_tempfile</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>108ms</td>
<td>169ms</td>
<td>Pod::Perldoc::find_good_formatter_class</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>82.9ms</td>
<td>83.6ms</td>
<td>Pod::Perldoc::opt_o_with</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>76.6ms</td>
<td>77.0ms</td>
<td>Pod::Perldoc::maybe_diddle_INC</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>60.3s</td>
<td>110ms</td>
<td>base::import</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>54.5ms</td>
<td>54.5ms</td>
<td>XML::Loader::load</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>52.1ms</td>
<td>52.1ms</td>
<td>Pod::Perldoc::CORE::sleep(xsub)</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>49.4ms</td>
<td>49.4ms</td>
<td>base::has_version</td>
</tr>
<tr>
<td>202</td>
<td>4</td>
<td>2</td>
<td>22.5ms</td>
<td>22.5ms</td>
<td>Pod::Perldoc::CORE::readline(xsub)</td>
</tr>
<tr>
<td>189</td>
<td>1</td>
<td>2</td>
<td>4.4ms</td>
<td>4.4ms</td>
<td>Pod::Perldoc::CORE::print(xsub)</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>9</td>
<td>2.5ms</td>
<td>2.5ms</td>
<td>File::Spec::Unix::as_heavy</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.18ms</td>
<td>1.70ms</td>
<td>File::Spec::Unix::import</td>
</tr>
</tbody>
</table>

See all 407 subroutines
Exclusive vs. Inclusive

• Exclusive Time is best for Bottom Up
  - Detail of time spent “in the code of this sub”
  - Where the time actually gets spent
  - Useful for localized (peephole) optimisation

• Inclusive Time is best for Top Down
  - Overview of time spent “in and below this sub”
  - Useful to prioritize structural optimizations
<table>
<thead>
<tr>
<th>Line</th>
<th>Statements</th>
<th>Time on line</th>
<th>Calls</th>
<th>Time in subs</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1294</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1295</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1296</td>
<td>1</td>
<td>2µs</td>
<td>1</td>
<td>447ms</td>
<td></td>
</tr>
<tr>
<td>1297</td>
<td>1</td>
<td>28µs</td>
<td>2</td>
<td>447ms</td>
<td></td>
</tr>
</tbody>
</table>

```
sub find {
    my $wanted = shift;
    _find_opt(wrap_wanted($wanted), @_);
    # spent 447ms making 1 call to File::Find::_find_opt
    # spent 18µs making 1 call to File::Find::wrap_wanted
}
```
<table>
<thead>
<tr>
<th>Line</th>
<th>Statements</th>
<th>Time on line</th>
<th>Calls</th>
<th>Time in subs</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1294</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1295</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1296</td>
<td>1</td>
<td>2µs</td>
<td>2</td>
<td>447ms</td>
<td></td>
</tr>
<tr>
<td>1297</td>
<td>1</td>
<td>28µs</td>
<td>2</td>
<td>447ms</td>
<td></td>
</tr>
<tr>
<td>1298</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1299</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Timings for each location calling into, or out of, the subroutine

Overall time spent in and below this sub (in + below)

Color coding based on Median Average Deviation relative to rest of this file

Timings for each location calling into, or out of, the subroutine

# spent 447ms (39µs+447) within File::Find::find which was called once (39µs+447ms) by main::RUNTIME at line 60 of demo/demo-code.pl

sub find {
    my $wanted = shift;
    _find_opt(wrap_wanted($wanted), @);
    # spent 447ms making 1 call to File::Find::_find_opt
    # spent 18µs making 1 call to File::Find::wrap_wanted
Boxes represent time spent in a subroutine. Coloring represents packages. Click to drill-down into package hierarchy.

_PPI::Lexer::_lex_structure

Called 246 times from 1 place in 1 file
Exclusive time: 36.5ms, 0.56%
Inclusive time: 504ms, 7.68%
Recursion: max depth 5, recursive inclusive time 262ms
Boxes represent subroutines
Colors only used to show packages (and aren’t pretty yet)

Hover over box to see details
Click to drill-down one level in package hierarchy
Calls between packages
Calls to/from/within package

[Diagram showing function calls and dependencies]
Let’s take a look...
Optimizing
Hints & Tips
Avoid 2!

Take care comparing code fragments!
Edge-effects at loop and scope boundaries
Time includes time *getting to* next statement
Avoid My Examples!

- Do your *own* testing
- With your *own* perl binary
- On your *own* hardware
Phase 0

Before you start
DON’T DO IT!
“The First Rule of Program Optimization: Don't do it.

The Second Rule of Program Optimization (for experts only!): Don't do it yet.”

- Michael A. Jackson
Why not?
“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.”

- W.A. Wulf
“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”

- Donald Knuth
“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”

- Donald Knuth
How?
“Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you have proven that's where the bottleneck is.”

- Rob Pike
“Measure twice, cut once.”

- Old Carpenter’s Maxim
Phase 1

Low Hanging Fruit
Low Hanging Fruit

1. Profile code running representative workload.
2. Look at Exclusive Time of subroutines.
3. Do they look reasonable?
4. Examine worst offenders.
5. Fix only simple local problems.
6. Profile again.
7. Fast enough? Then STOP!
8. Rinse and repeat once or twice, then move on.
“Simple Local Fixes”

Changes unlikely to introduce bugs
Move invariant expressions out of loops
Avoid repeated chains
  ->of->accessors(...);
Avoid repeated chains
  ->of->accessors(...);

Use a temporary variable
Use faster accessors

Class::Accessor
  -> Class::Accessor::Fast
  --> Class::Accessor::Faster
  ---› Class::Accessor::Fast::XS
Avoid calling subs that don’t do anything!

```perl
my $unused_variable = $self->get_foo;

my $is_logging = $log->info(...);
while (...) {
    $log->info(...) if $is_logging;
    ...
}
```
Exit subs and loops early
Delay initializations

return if not ...a cheap test...;
return if not ...a more expensive test...;
my $foo = ...initializations...;
...body of subroutine...
Fix silly code

- return exists $nav_type{$country}{$key}
-     ? $nav_type{$country}{$key}
-     : undef;
+ return $nav_type{$country}{$key};;
Beware pathological regular expressions

Devel::NYTProf shows regular expression opcodes
Avoid unpacking args in very hot subs

```perl
sub foo { shift->delegate(@_) }

sub bar {
    return shift->{bar} unless @_;  
    return $_[0]->{bar} = $_[1];
}
```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>200000</td>
<td>821ms</td>
<td># spent 720ms within main::square1 which was called 100000 times</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># 100000 times (720ms+0s) by main::RUNTIME at line 8, avg 7μs/call</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sub square1 { my ($n) = @_; return $n * $n; }</td>
</tr>
<tr>
<td>4</td>
<td>200000</td>
<td>818ms</td>
<td># spent 719ms within main::square2 which was called 100000 times</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># 100000 times (719ms+0s) by main::RUNTIME at line 9, avg 7μs/call</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sub square2 { my $n = shift; return $n * $n; }</td>
</tr>
<tr>
<td>5</td>
<td>100000</td>
<td>687ms</td>
<td># spent 592ms within main::square3 which was called 100000 times</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># 100000 times (592ms+0s) by main::RUNTIME at line 10, avg 6μs/call</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sub square3 { return $<em>[0] * $</em>[0]; }</td>
</tr>
</tbody>
</table>
Retest.

Fast enough?

STOP!

Put the profiler down and walk away
Phase 2

Deeper Changes
Profile with a known workload

E.g., 1000 identical requests
Check Inclusive Times
(especially top-level subs)

Reasonable percentage for the workload?
Check subroutine call counts

Reasonable for the workload?
Add caching *if appropriate* to reduce calls

Remember invalidation!
Walk up call chain to find good spots for caching

Remember invalidation!
Creating many objects that don’t get used?

Lightweight proxies

e.g. DateTimeX::Lite
Retest.

Fast enough?

STOP!

Put the profiler down and walk away
Phase 3

Structural Changes
Push loops down

- $object->walk($_) for @dogs;

+ $object->walk_these(@dogs);
Avoid needless closures

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>800ns</td>
<td>our $var;</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>18μs</td>
<td>for my $i (1..100_000) {</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 7 | 300000 | 258ms | my $code1 = \\
| 8 | 300000 | 562ms | foo; sub foo { $var; 1 } |
| 9 | 300000 | 518ms | my $code2 = sub { $var; 1 }; |
| 10 |   |   |   |
| 11 | 100000 | 143ms | my $code3 = sub { $i; 1 }; |
| 12 | 100000 | 102ms | $code1->(); |
| 13 | 100000 | 104ms | $code2->(); |
| 14 | 100000 | 654ms | $code3->(); |
| 15 |   |   | 1; |
|   |   |   | } |
Change the data structure

hashes $\leftrightarrow$ arrays
Change the algorithm

What’s the “Big O”?  
$O(n^2)$ or $O(\log n)$ or ...
Rewrite hot-spots in C

Inline::C
Small changes add up!

“I achieved my fast times by multitudes of 1% reductions”

- Bill Raymond
Questions?

Tim.Bunce@pobox.com
http://blog.timbunce.org
@timbunce on twitter