Devel::NYTProf

Perl Source Code Profiler

Tim Bunce - July 2009

Screencast available at
http://blog.timbunce.org/tag/nytprof/

Devel::DProf

- Oldest Perl Profiler —1995
- Design flaws make it practically useless on modern systems
- Limited to 0.01 second resolution even for realtime measurements!

Devel::DProf *Is Broken*

```
$ 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
   %Time ExclSec CumulS #Calls sec/call Csec/c  Name
    9.26   0.010  0.010      1   0.0100 0.0100  main::s76
    9.26   0.010  0.010      1   0.0100 0.0100  main::s323
    9.26   0.010  0.010      1   0.0100 0.0100  main::s626
    9.26   0.010  0.010      1   0.0100 0.0100  main::s936
    0.00       - -0.000      1        -      -  main::s77
    0.00       - -0.000      1        -      -  main::s82
```

Lots of Perl Profilers

- Take your pick...

Devel::DProf	1995	Subroutine
Devel::SmallProf	1997	Line
Devel::AutoProfiler	2002	Subroutine
Devel::Profiler	2002	*Subroutine*
Devel::Profile	2003	*Subroutine*
Devel::FastProf	2005	Line
Devel::DProfLB	2006	*Subroutine*
Devel::WxProf	2008	*Subroutine*
Devel::Profit	2008	Line
Devel::NYTProf	2008	Line & Subroutine

Evolution

Devel::DProf	1995	Subroutine
Devel::SmallProf	1997	Line
Devel::AutoProfiler	2002	Subroutine
Devel::Profiler	2002	Subroutine
Devel::Profile	2003	Subroutine
Devel::FastProf	2005	Line
Devel::DProfLB	2006	Subroutine
Devel::WxProf	2008	Subroutine
Devel::Profit	2008	Line
Devel::NYTProf v1	2008	Line
Devel::NYTProf v2	2008	Line & Subroutine

...plus lots of innovations!

What To Measure?

	CPU Time	Real Time
Subroutines	?	?
Statements	?	?

CPU Time *vs* Real Time

- CPU time
  - Very poor resolution (0.01s) on many systems
  - Not (much) affected by load on system
  - Doesn't include time spent waiting for i/o etc.
- Real time
  - High resolution: microseconds or better
  - Is affected by load on system
  - Includes time spent waiting

Sub *vs* Line

- 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
  - No insight into where time spent *within* large subs
  - Doesn't measure code outside of a sub

Sub *vs* Line

- 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 (want time per sub)
  - Hard to get overall subroutine times

Devel::NYTProf

v1 *Innovations*

- Fork of Devel::FastProf by Adam Kaplan
  - 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*

v2 *Innovations*

- Dual Profilers!
  - Is a statement profiler
  - *and* a subroutine profiler
  - At the same time!

v2 *Innovations*

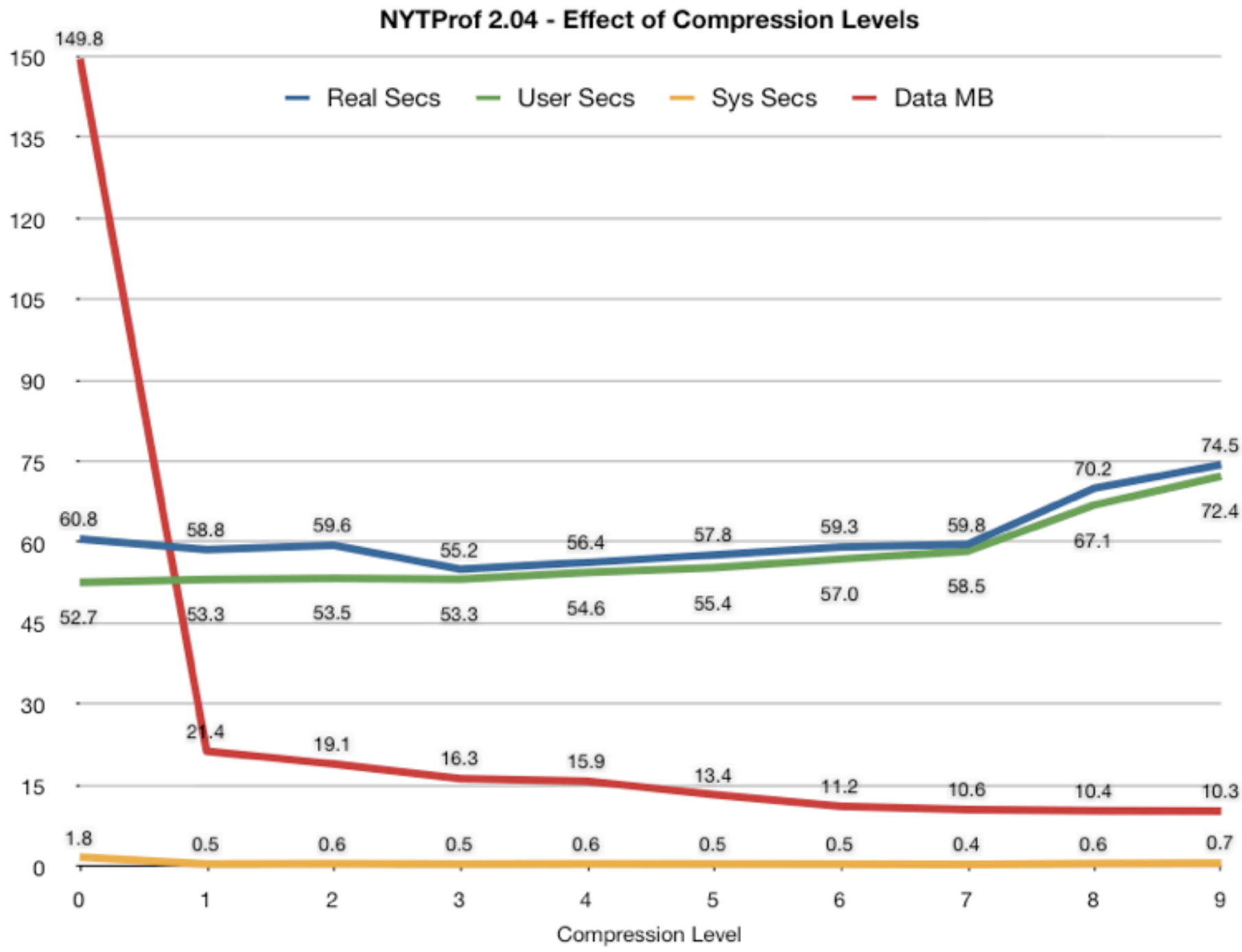
- Subroutine profiler
  - tracks time *per calling location*
  - even for xsubs
  - calculates exclusive time on-the-fly
  - discounts overhead of statement profiler
  - immune from funky control flow
  - in memory, writes to file at end
  - *extremely fast*

v2 *Innovations*

- Statement profiler gives correct timing after *leave* ops
  - *unlike* previous statement profilers...
  - 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

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


NYTProf 2.04 - Effect of Compression Levels
Real Secs
User Secs
Sys Secs
Data MB
150
135
120
105
90
75
60
45
30
15
0
149.8
60.8
58.8
59.6
55.2
56.4
57.8
59.3
59.8
70.2
74.5
52.7
53.3
53.5
53.3
54.6
55.4
57.0
58.5
67.1
72.4
21.4
19.1
16.3
15.9
13.4
11.2
10.6
10.4
10.3
1.8
0.5
0.6
0.5
0.6
0.5
0.5
0.4
0.6
0.7
0
1
2
3
4
5
6
7
8
9
Compression Level

Profiling Performance

	Time	Size
Perl	x 1	-
SmallProf	x 22	-
FastProf	x 6.3	42,927KB
NYTProf	x 3.9	11,174KB
+ blocks=0	x 3.5	9,628KB
+ stmts=0	x 2.5*	205KB
DProf	x 4.9	60,736KB

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

Running NYTProf

```
perl -d:NYTProf ...

perl -MDevel::NYTProf ...

PERL5OPT=-d:NYTProf

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

Reporting NYTProf

- CSV - old, limited, dull

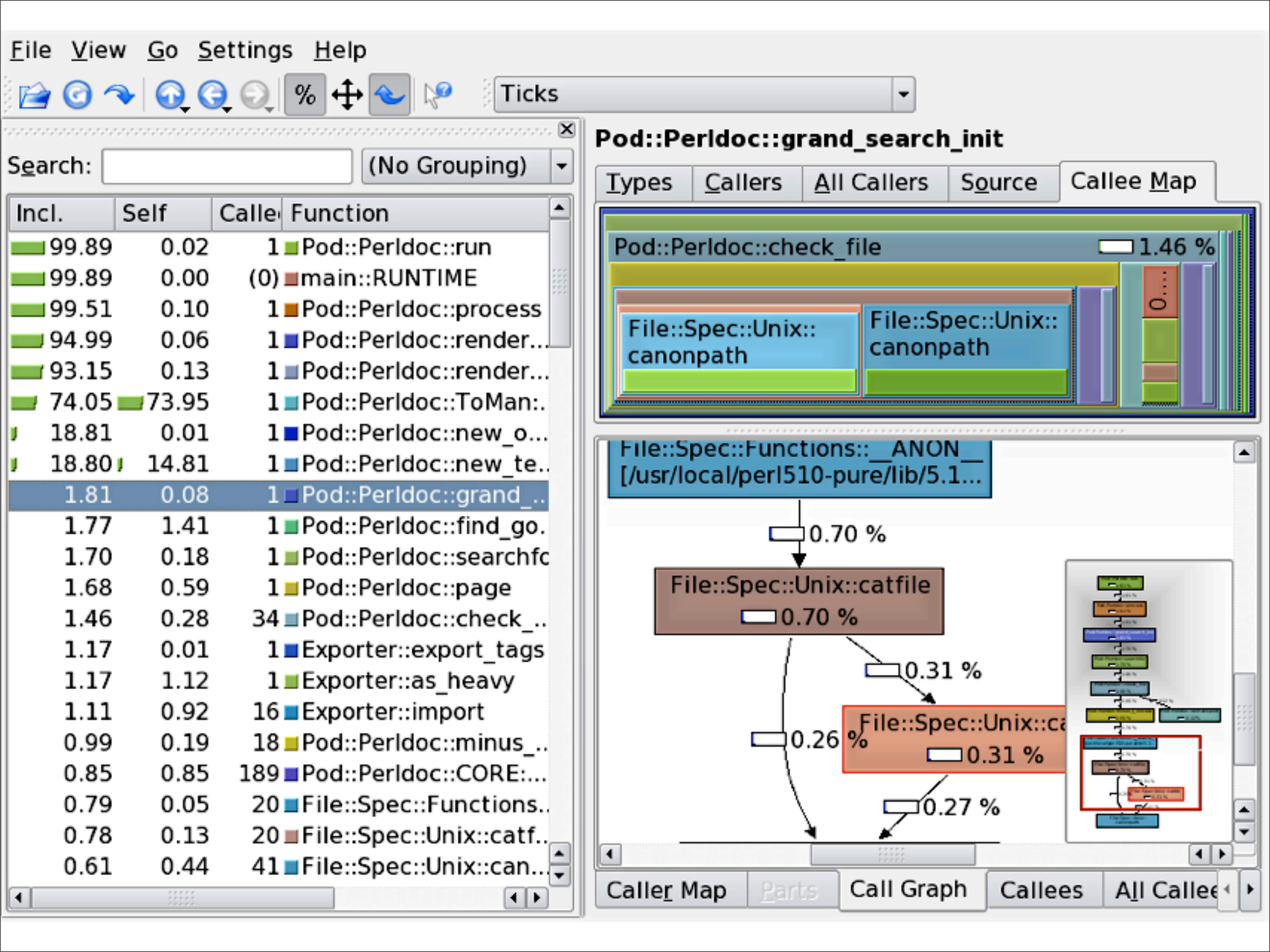
```
$ nytprofcsv
```

```
# Format: time,calls,time/call,code
0,0,0,sub foo {
0.000002,2,0.00001,print "in sub foo\n";
0.000004,2,0.00002,bar();
0,0,0,}
0,0,0,
```

Reporting NYTProf

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

```
$ nytprofcg   # generates nytprof.callgraph
$ kcachegrind # load the file via the gui
```


File View Go Settings Help
Ticks
Search:
(No Grouping)
Incl. Self Calle Function
99.89 0.02 1 Pod::Perldoc::run
99.89 0.00 (0) main::RUNTIME
99.51 0.10 1 Pod::Perldoc::process
94.99 0.06 1 Pod::Perldoc::render...
93.15 0.13 1 Pod::Perldoc::render...
74.05 73.95 1 Pod::Perldoc::ToMan:..
18.81 0.01 1 Pod::Perldoc::new_o...
18.80 14.81 1 Pod::Perldoc::new_te..
1.81 0.08 1 Pod::Perldoc::grand_..
1.77 1.41 1 Pod::Perldoc::find_go.
1.70 0.18 1 Pod::Perldoc::searchfc
1.68 0.59 1 Pod::Perldoc::page
1.46 0.28 34 Pod::Perldoc::check_..
1.17 0.01 1 Exporter::export_tags
1.17 1.12 1 Exporter::as_heavy
1.11 0.92 16 Exporter::import
0.99 0.19 18 Pod::Perldoc::minus_..
0.85 0.85 189 Pod::Perldoc::CORE:...
0.79 0.05 20 File::Spec::Functions..
0.78 0.13 20 File::Spec::Unix::catf..
0.61 0.44 41 File::Spec::Unix::can...
Pod::Perldoc::grand_search_init
Types Callers All Callers Source Callee Map
Pod::Perldoc::check_file
1.46 %
File::Spec::Unix::
canonpath
File::Spec::Unix::
canonpath
File::Spec::Functions::__ANON__
[/usr/local/perl510-pure/lib/5.1...
0.70 %
File::Spec::Unix::catfile
0.70 %
0.31 %
0.26 %
File::Spec::Unix::ca
0.31 %
0.27 %
Caller Map Parts Call Graph Callees All Callee

Reporting NYTProf

- 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
```

Performance Profile Index

For /usr/local/perl510-pure/bin/perldoc

Run on Wed Jul 22 00:04:40 2009

Reported on Wed Jul 22 00:04:48 2009

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.

Jump to file...

Top 15 Subroutines — ordered by exclusive time

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

See all 407 subroutines

Performance Profile Index

For /usr/local/perl510-pure/bin/perldoc

Run on Wed Jul 22 00:04:40 2009
Reported on Wed Jul 22 00:04:48 2009

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.

Jump to file...

Links to annotated source code

Top 15 Subroutines — ordered by exclusive time

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
1	1	1	1.87s	1.87s	Pod::Perldoc::ToMan::parse_from_file
1	1	1	637ms	750ms	Pod::Perldoc::new_tempfile
1	1	1	108ms	169ms	Pod::Perldoc::find_good_formatter_class
2	2	1	82.9ms	83.6ms	Pod::Perldoc::opt_o_with
1	1	1	76.6ms	77.0ms	Pod::Perldoc::maybe_diddle_INC
3	3	2	60.3ms	110ms	base::import
2	2	2	54.5ms	54.5ms	XSLoader::load
3	1	2	52.1ms	52.1ms	Pod::Perldoc::CORE:sleep(xsub)
3	1	1	49.4ms	49.4ms	base::has_version
202	4	2	22.5ms	22.5ms	Pod::Perldoc::CORE:readline(xsub)
189	1	2	4		doc::CORE:print(xsub)
1	1	1	2		ter::as_heavy
16	16	9	2		ter::import
1	1	1	1		doc:
41	3	1	1.18ms	1.70ms	File::Spec::Unix:

Link to sortable table of all subs

Timings for perl builtins

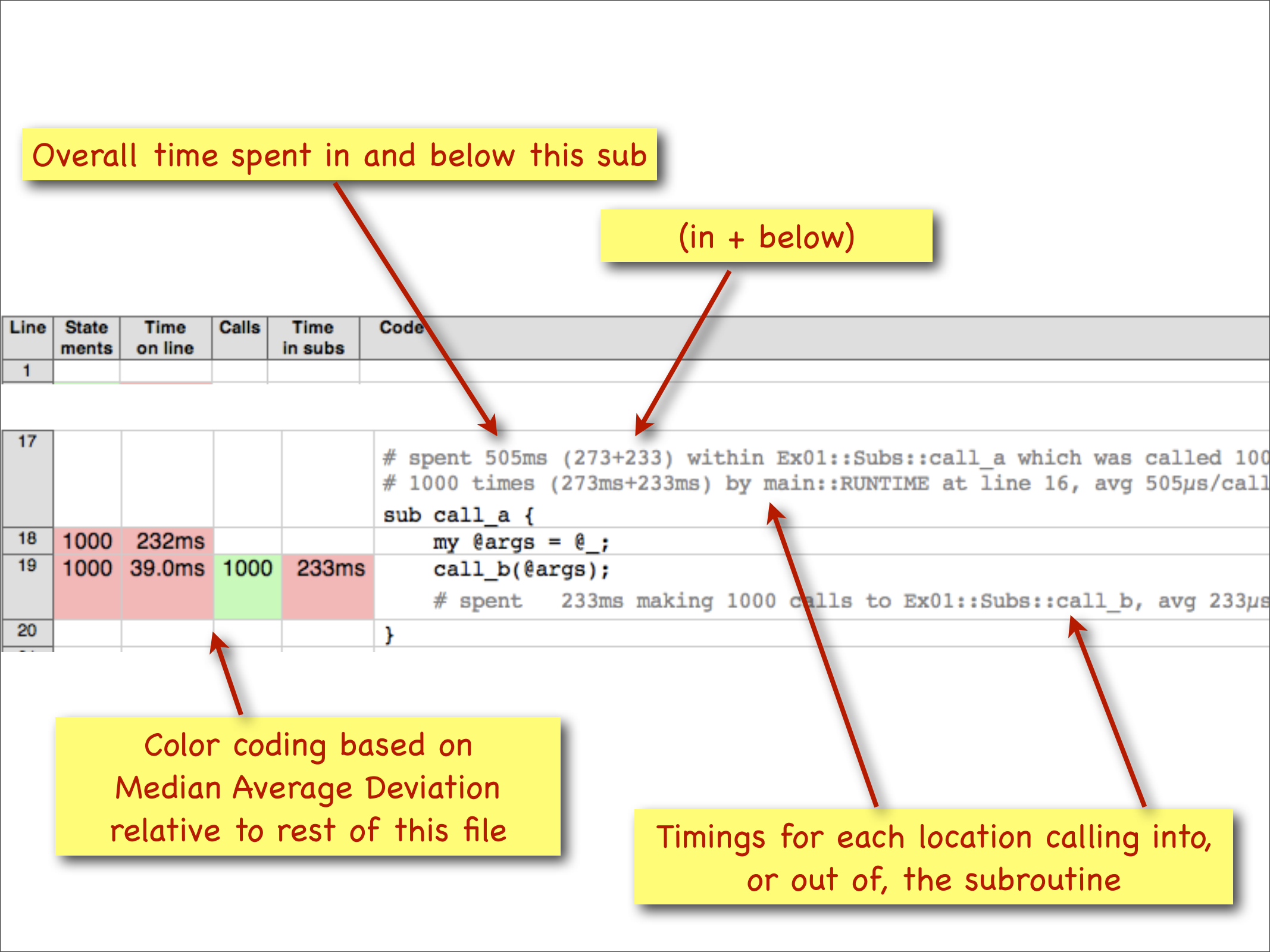
See all 407 subroutines

Exclusive *vs.* Inclusive

- Exclusive Time = Bottom up
  - Detail of time spent *"just here"*
  - Where the time *actually* gets spent
  - Useful for localized (peephole) optimisation

- Inclusive Time = Top down
  - Overview of time spent *"in and below"*
  - Useful to prioritize structural optimizations

Line	State ments	Time on line	Calls	Time in subs	Code
1					
17					# spent 505ms (273+233) within Ex01::Subs::call_a which was called 100 # 1000 times (273ms+233ms) by main::RUNTIME at line 16, avg 505µs/call sub call_a {
18	1000	232ms			my @args = @_;
19	1000	39.0ms	1000	233ms	call_b(@args); # spent 233ms making 1000 calls to Ex01::Subs::call_b, avg 233µs
20					}

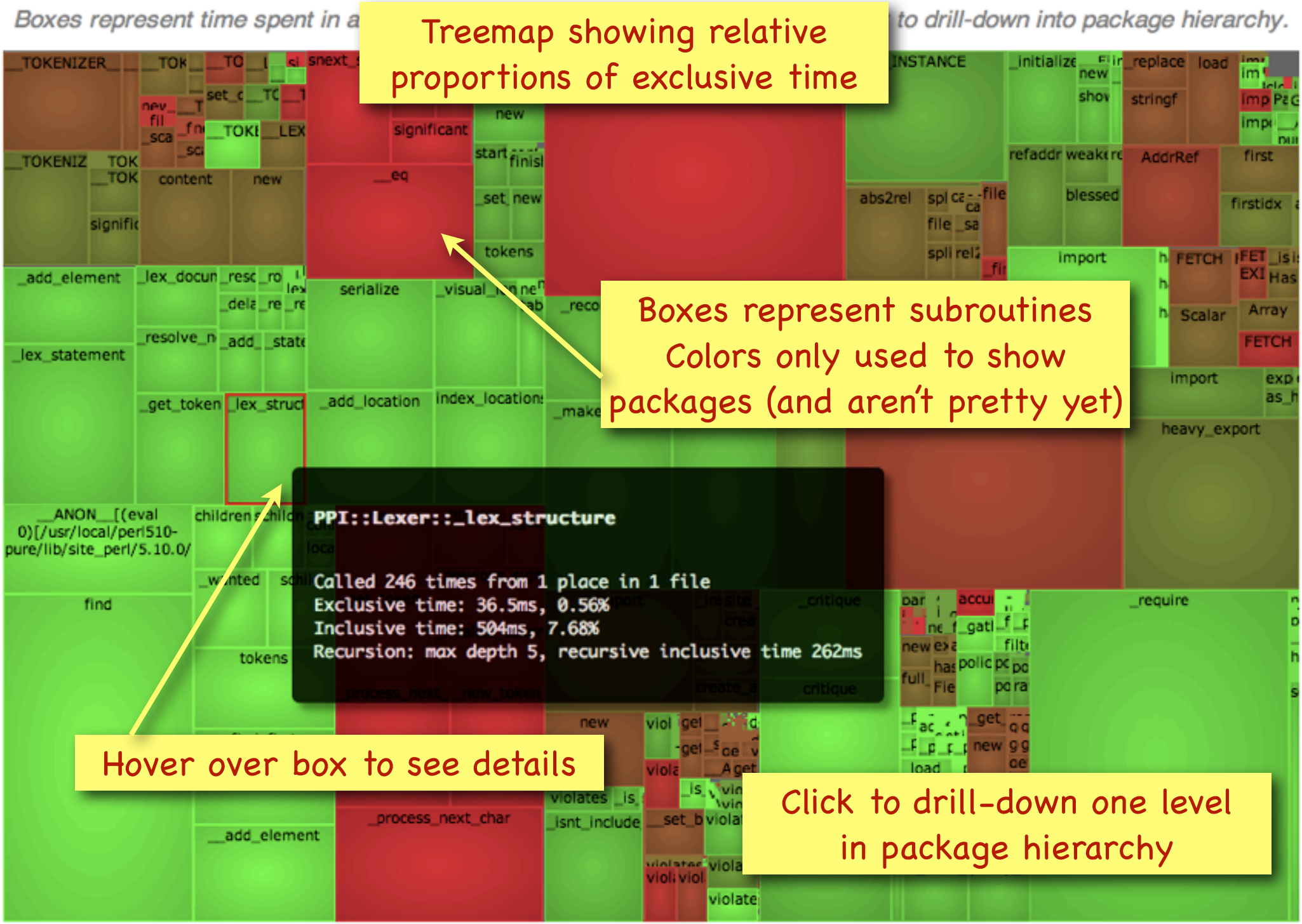


Line	State ments	Time on line	Calls	Time in subs	Code
1					

17					`# spent 505ms (273+233) within Ex01::Subs::call_a which was called 100`
					`# 1000 times (273ms+233ms) by main::RUNTIME at line 16, avg 505µs/call`
					`sub call_a {`

18	1000	232ms			`    my @args = @_;`
19	1000	39.0ms	1000	233ms	`    call_b(@args);` `    # spent   233ms making 1000 calls to Ex01::Subs::call_b, avg 233µs`
20					`}`

Boxes represent time spent in a subroutine. Coloring represents packages. Click to drill-down into package hierarchy.


Boxes represent time spent in a
to drill-down into package hierarchy.
Treemap showing relative proportions of exclusive time
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
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
__TOKENIZER__
content
new
significant
__eq
serialize
tokens
_add_element
_lex_statement
_get_token
_lex_struct
_add_location
index_locations
__ANON__[(eval 0)[/usr/local/perl510-pure/lib/site_perl/5.10.0/
find
_process_next_char
__add_element
_isnt_include
_require
heavy_export
import
Scalar
Array
FETCH
AddrRef
first
firstidx
stringf
blessed
abs2rel
_initialize
_replace
load
INSTANCE

Let's take a look...

Optimizing

Hints & Tips

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 Proverb

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(...)

Use a temporary variable

Use faster accessors

```
Class::Accessor
-> Class::Accessor::Fast
--> Class::Accessor::Faster
---> Class::XSAccessor
```

Avoid calling subs that don't do anything!

```
my $unsed_variable = $self->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

`NYTPROF=slowops=2`

Avoid unpacking args in very hot subs

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

sub bar {
    return shift->{bar} unless @_;
    return $_[0]->{bar} = $_[1];
}
```

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);
```

Change the data structure

hashes <–> arrays

Change the algorithm

What's the "Big O"?
$O(n^2)$ or $O(\log n)$ or ...

Rewrite hot-spots in C

`Inline::C`

It all adds up!

"I achieved my fast times by multitudes of 1% reductions"

- Bill Raymond

Questions?

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