

## NAME

perlperf - Perl Performance and Optimization Techniques

## DESCRIPTION

This is an introduction to the use of performance and optimization techniques which can be used with particular reference to perl programs. While many perl developers have come from other languages, and can use their prior knowledge where appropriate, there are many other people who might benefit from a few perl specific pointers. If you want the condensed version, perhaps the best advice comes from the renowned Japanese Samurai, Miyamoto Musashi, who said:

```
"Do Not Engage in Useless Activity"
```

in 1645.

## OVERVIEW

Perhaps the most common mistake programmers make is to attempt to optimize their code before a program actually does anything useful - this is a bad idea. There's no point in having an extremely fast program that doesn't work. The first job is to get a program to *correctly* do something **useful**, (not to mention ensuring the test suite is fully functional), and only then to consider optimizing it. Having decided to optimize existing working code, there are several simple but essential steps to consider which are intrinsic to any optimization process.

## ONE STEP SIDEWAYS

Firstly, you need to establish a baseline time for the existing code, which timing needs to be reliable and repeatable. You'll probably want to use the `Benchmark` or `Devel::DProf` modules, or something similar, for this step, or perhaps the unix system `time` utility, whichever is appropriate. See the base of this document for a longer list of benchmarking and profiling modules, and recommended further reading.

## ONE STEP FORWARD

Next, having examined the program for *hot spots*, (places where the code seems to run slowly), change the code with the intention of making it run faster. Using version control software, like `subversion`, will ensure no changes are irreversible. It's too easy to fiddle here and fiddle there - don't change too much at any one time or you might not discover which piece of code **really** was the slow bit.

## ANOTHER STEP SIDEWAYS

It's not enough to say: "that will make it run faster", you have to check it. Rerun the code under control of the benchmarking or profiling modules, from the first step above, and check that the new code executed the **same task** in *less time*. Save your work and repeat...

## GENERAL GUIDELINES

The critical thing when considering performance is to remember there is no such thing as a `Golden Bullet`, which is why there are no rules, only guidelines.

It is clear that inline code is going to be faster than subroutine or method calls, because there is less overhead, but this approach has the disadvantage of being less maintainable and comes at the cost of greater memory usage - there is no such thing as a free lunch. If you are searching for an element in a list, it can be more efficient to store the data in a hash structure, and then simply look to see whether the key is defined, rather than to loop through the entire array using `grep()` for instance. `substr()` may be (a lot) faster than `grep()` but not as flexible, so you have another trade-off to access. Your code may contain a line which takes 0.01 of a second to execute which if you call it 1,000 times, quite likely in a program parsing even medium sized files for instance, you already have a 10 second delay, in just one single code location, and if you call that line 100,000 times, your entire program will slow down to an unbearable crawl.

Using a subroutine as part of your sort is a powerful way to get exactly what you want, but will usually be slower than the built-in *alphabetic* `cmp` and *numeric* `<=>` sort operators. It is possible to make multiple passes over your data, building indices to make the upcoming sort more efficient, and to use what is known as the *OM* (Orcish Maneuver) to cache the sort keys in advance. The cache lookup, while a good idea, can itself be a source of slowdown by enforcing a double pass over the data - once to setup the cache, and once to sort the data. Using `pack()` to extract the required sort key into a consistent string can be an efficient way to build a single string to compare, instead of using multiple sort keys, which makes it possible to use the standard, written in *c* and fast, `perl sort()` function on the output, and is the basis of the *GRT* (Guttman Rossler Transform). Some string combinations can slow the *GRT* down, by just being too plain complex for it's own good.

For applications using database backends, the standard *DBIx* namespace has tries to help with keeping things nippy, not least because it tries to *not* query the database until the latest possible moment, but always read the docs which come with your choice of libraries. Among the many issues facing developers dealing with databases should remain aware of is to always use *SQL* placeholders and to consider pre-fetching data sets when this might prove advantageous. Splitting up a large file by assigning multiple processes to parsing a single file, using *say* *POE*, *threads* or *fork* can also be a useful way of optimizing your usage of the available *CPU* resources, though this technique is fraught with concurrency issues and demands high attention to detail.

Every case has a specific application and one or more exceptions, and there is no replacement for running a few tests and finding out which method works best for your particular environment, this is why writing optimal code is not an exact science, and why we love using Perl so much - *TMTOWTDI*.

## BENCHMARKS

Here are a few examples to demonstrate usage of Perl's benchmarking tools.

### Assigning and Dereferencing Variables.

I'm sure most of us have seen code which looks like, (or worse than), this:

```
if ( $obj->{_ref}->{_myscore} >= $obj->{_ref}->{_yourscore} ) {
    ...
}
```

This sort of code can be a real eyesore to read, as well as being very sensitive to typos, and it's much clearer to dereference the variable explicitly. We're side-stepping the issue of working with object-oriented programming techniques to encapsulate variable access via methods, only accessible through an object. Here we're just discussing the technical implementation of choice, and whether this has an effect on performance. We can see whether this dereferencing operation, has any overhead by putting comparative code in a file and running a *Benchmark* test.

# dereference

```
#!/usr/bin/perl

use strict;
use warnings;

use Benchmark;

my $ref = {
    'ref' => {
        _myscore => '100 + 1',
        _yourscore => '102 - 1',
    },
};
```

```

timethese(1000000, {
    'direct' => sub {
        my $x = $ref->{ref}->{_myscore} . $ref->{ref}->{_yourscore}
    },
    'dereference' => sub {
        my $ref = $ref->{ref};
        my $myscore = $ref->{_myscore};
        my $yourscore = $ref->{_yourscore};
        my $x = $myscore . $yourscore;
    },
});

```

It's essential to run any timing measurements a sufficient number of times so the numbers settle on a numerical average, otherwise each run will naturally fluctuate due to variations in the environment, to reduce the effect of contention for CPU resources and network bandwidth for instance. Running the above code for one million iterations, we can take a look at the report output by the `Benchmark` module, to see which approach is the most effective.

```
$> perl dereference
```

```

Benchmark: timing 1000000 iterations of dereference, direct...
dereference:  2 wallclock secs ( 1.59 usr +  0.00 sys =  1.59 CPU) @
628930.82/s (n=1000000)
direct:  1 wallclock secs ( 1.20 usr +  0.00 sys =  1.20 CPU) @
833333.33/s (n=1000000)

```

The difference is clear to see and the dereferencing approach is slower. While it managed to execute an average of 628,930 times a second during our test, the direct approach managed to run an additional 204,403 times, unfortunately. Unfortunately, because there are many examples of code written using the multiple layer direct variable access, and it's usually horrible. It is, however, miniscully faster. The question remains whether the minute gain is actually worth the eyestrain, or the loss of maintainability.

## Search and replace or tr

If we have a string which needs to be modified, while a regex will almost always be much more flexible, `tr`, an oft underused tool, can still be a useful. One scenario might be replace all vowels with another character. The regex solution might look like this:

```
$str =~ s/[aeiou]/x/g
```

The `tr` alternative might look like this:

```
$str =~ tr/aeiou/xxxxx/
```

We can put that into a test file which we can run to check which approach is the fastest, using a global `$STR` variable to assign to the `my $str` variable so as to avoid perl trying to optimize any of the work away by noticing it's assigned only the once.

```
# regex-transliterate
```

```
#!/usr/bin/perl
```

```
use strict;
use warnings;
```

```
use Benchmark;

my $STR = "$$-this and that";

timethese( 1000000, {
    'sr' => sub { my $str = $STR; $str =~ s/[aeiou]/x/g; return
$str; },
    'tr' => sub { my $str = $STR; $str =~ tr/aeiou/xxxxx/; return
$str; },
});
```

Running the code gives us our results:

```
$> perl regex-transliterate

Benchmark: timing 1000000 iterations of sr, tr...
   sr:  2 wallclock secs ( 1.19 usr +  0.00 sys =  1.19 CPU) @
840336.13/s (n=1000000)
   tr:  0 wallclock secs ( 0.49 usr +  0.00 sys =  0.49 CPU) @
2040816.33/s (n=1000000)
```

The `tr` version is a clear winner. One solution is flexible, the other is fast - and it's appropriately the programmers choice which to use in the circumstances.

Check the Benchmark docs for further useful techniques.

## PROFILING TOOLS

A slightly larger piece of code will provide something on which a profiler can produce more extensive reporting statistics. This example uses the simplistic `wordmatch` program which parses a given input file and spews out a short report on the contents.

```
# wordmatch
```

```
#!/usr/bin/perl
```

```
use strict;
use warnings;
```

```
=head1 NAME
```

```
filewords - word analysis of input file
```

```
=head1 SYNOPSIS
```

```
filewords -f inputfilename [-d]
```

```
=head1 DESCRIPTION
```

```
This program parses the given filename, specified with C<-f>, and displays a simple analysis of the words found therein. Use the C<-d> switch to enable debugging messages.
```

```
=cut

use FileHandle;
use Getopt::Long;

my $debug    = 0;
my $file     = '';

my $result = GetOptions (
    'debug'      => \$debug,
    'file=s'     => \$file,
);
die("invalid args") unless $result;

unless ( -f $file ) {
    die("Usage: $0 -f filename [-d]");
}
my $FH = FileHandle->new("< $file") or die("unable to open file($file):
$!");

my $i_LINES = 0;
my $i_WORDS = 0;
my %count   = ();

my @lines = <$FH>;
foreach my $line ( @lines ) {
    $i_LINES++;
    $line =~ s/\n//;
    my @words = split(/ +/, $line);
    my $i_words = scalar(@words);
    $i_WORDS = $i_WORDS + $i_words;
    debug("line: $i_LINES supplying $i_words words: @words");
    my $i_word = 0;
    foreach my $word ( @words ) {
        $i_word++;
        $count{$i_LINES}{spec} += matches($i_word, $word,
'[^a-zA-Z0-9]');
        $count{$i_LINES}{only} += matches($i_word, $word,
'^[^a-zA-Z0-9]+$');
        $count{$i_LINES}{cons} += matches($i_word, $word,
'^([?i:bcdfghjklmnpqrstvwxyz])+');
        $count{$i_LINES}{vows} += matches($i_word, $word,
'^([?i:aeiou])+');
        $count{$i_LINES}{caps} += matches($i_word, $word,
'^[(A-Z)]+$');
    }
}

print report( %count );

sub matches {
    my $i_wd = shift;
    my $word = shift;
```

```
my $regex = shift;
my $has = 0;

if ( $word =~ /($regex)/ ) {
    $has++ if $1;
}

debug("word: $i_wd " . ($has ? 'matches' : 'does not match') . " chars:
/$regex/");

return $has;
}

sub report {
    my %report = @_;
    my %rep;

    foreach my $line ( keys %report ) {
        foreach my $key ( keys %{ $report{$line} } ) {
            $rep{$key} += $report{$line}{$key};
        }
    }

    my $report = qq|
$0 report for $file:
lines in file: $i_LINES
words in file: $i_WORDS
words with special (non-word) characters: $i_spec
words with only special (non-word) characters: $i_only
words with only consonants: $i_cons
words with only capital letters: $i_caps
words with only vowels: $i_vows
|;

    return $report;
}

sub debug {
    my $message = shift;

    if ( $debug ) {
        print STDERR "DBG: $message\n";
    }
}

exit 0;
```

### Devel::DProf

This venerable module has been the de-facto standard for Perl code profiling for more than a decade, but has been replaced by a number of other modules which have brought us back to the 21st century. Although you're recommended to evaluate your tool from the several mentioned here and from the CPAN list at the base of this document, (and currently *Devel::NYTProf* seems to be the weapon of

choice - see below), we'll take a quick look at the output from `Devel::DProf` first, to set a baseline for Perl profiling tools. Run the above program under the control of `Devel::DProf` by using the `-d` switch on the command-line.

```
$> perl -d:DProf wordmatch -f perl5db.pl

<...multiple lines snipped...>

wordmatch report for perl5db.pl:
lines in file: 9428
words in file: 50243
words with special (non-word) characters: 20480
words with only special (non-word) characters: 7790
words with only consonants: 4801
words with only capital letters: 1316
words with only vowels: 1701
```

`Devel::DProf` produces a special file, called `tmon.out` by default, and this file is read by the `dprofpp` program, which is already installed as part of the `Devel::DProf` distribution. If you call `dprofpp` with no options, it will read the `tmon.out` file in the current directory and produce a human readable statistics report of the run of your program. Note that this may take a little time.

```
$> dprofpp

Total Elapsed Time = 2.951677 Seconds
  User+System Time = 2.871677 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c Name
102.   2.945   3.003 251215  0.0000 0.0000 main::matches
 2.40   0.069   0.069 260643  0.0000 0.0000 main::debug
 1.74   0.050   0.050    1  0.0500 0.0500 main::report
 1.04   0.030   0.049    4  0.0075 0.0123 main::BEGIN
 0.35   0.010   0.010    3  0.0033 0.0033 Exporter::as_heavy
 0.35   0.010   0.010    7  0.0014 0.0014 IO::File::BEGIN
 0.00    - -0.000    1    -    - Getopt::Long::FindOption
 0.00    - -0.000    1    -    - Symbol::BEGIN
 0.00    - -0.000    1    -    - Fcntl::BEGIN
 0.00    - -0.000    1    -    - Fcntl::bootstrap
 0.00    - -0.000    1    -    - warnings::BEGIN
 0.00    - -0.000    1    -    - IO::bootstrap
 0.00    - -0.000    1    -    -
Getopt::Long::ConfigDefaults
 0.00    - -0.000    1    -    - Getopt::Long::Configure
 0.00    - -0.000    1    -    - Symbol::gensym
```

`dprofpp` will produce some quite detailed reporting on the activity of the `wordmatch` program. The wallclock, user and system, times are at the top of the analysis, and after this are the main columns defining which define the report. Check the `dprofpp` docs for details of the many options it supports.

See also `Apache::DProf` which hooks `Devel::DProf` into `mod_perl`.

## Devel::Profiler

Let's take a look at the same program using a different profiler: `Devel::Profiler`, a drop-in Perl-only replacement for `Devel::DProf`. The usage is very slightly different in that instead of using the special `-d:` flag, you pull `Devel::Profiler` in directly as a module using `-M`.

```
$> perl -MDevel::Profiler wordmatch -f perl5db.pl
```

```
<...multiple lines snipped...>
```

```
wordmatch report for perl5db.pl:
lines in file: 9428
words in file: 50243
words with special (non-word) characters: 20480
words with only special (non-word) characters: 7790
words with only consonants: 4801
words with only capital letters: 1316
words with only vowels: 1701
```

Devel::Profiler generates a tmon.out file which is compatible with the dprofpp program, thus saving the construction of a dedicated statistics reader program. dprofpp usage is therefore identical to the above example.

```
$> dprofpp
```

```
Total Elapsed Time = 20.984 Seconds
  User+System Time = 19.981 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c Name
 49.0  9.792 14.509 251215  0.0000 0.0001 main::matches
 24.4  4.887  4.887 260643  0.0000 0.0000 main::debug
  0.25 0.049  0.049     1  0.0490 0.0490 main::report
  0.00 0.000  0.000     1  0.0000 0.0000 Getopt::Long::GetOptions
  0.00 0.000  0.000     2  0.0000 0.0000
Getopt::Long::ParseOptionSpec
  0.00 0.000  0.000     1  0.0000 0.0000 Getopt::Long::FindOption
  0.00 0.000  0.000     1  0.0000 0.0000 IO::File::new
  0.00 0.000  0.000     1  0.0000 0.0000 IO::Handle::new
  0.00 0.000  0.000     1  0.0000 0.0000 Symbol::gensym
  0.00 0.000  0.000     1  0.0000 0.0000 IO::File::open
```

Interestingly we get slightly different results, which is mostly because the algorithm which generates the report is different, even though the output file format was allegedly identical. The elapsed, user and system times are clearly showing the time it took for Devel::Profiler to execute it's own run, but the column listings feel more accurate somehow than the ones we had earlier from Devel::DProf. The 102% figure has disappeared, for example. This is where we have to use the tools at our disposal, and recognise their pros and cons, before using them. Interestingly, the numbers of calls for each subroutine are identical in the two reports, it's the percentages which differ. As the author of Devel::Profiler writes:

```
...running HTML::Template's test suite under Devel::DProf shows
output()
taking NO time but Devel::Profiler shows around 10% of the time is in
output().
I don't know which to trust but my gut tells me something is wrong with
Devel::DProf. HTML::Template::output() is a big routine that's called
for
every test. Either way, something needs fixing.
```

YMMV.

See also `Devel::Apache::Profiler` which hooks `Devel::Profiler` into `mod_perl`.

## Devel::SmallProf

The `Devel::SmallProf` profiler examines the runtime of your Perl program and produces a line-by-line listing to show how many times each line was called, and how long each line took to execute. It is called by supplying the familiar `-d` flag to Perl at runtime.

```
$> perl -d:SmallProf wordmatch -f perl5db.pl
```

```
<...multiple lines snipped...>
```

```
wordmatch report for perl5db.pl:
lines in file: 9428
words in file: 50243
words with special (non-word) characters: 20480
words with only special (non-word) characters: 7790
words with only consonants: 4801
words with only capital letters: 1316
words with only vowels: 1701
```

`Devel::SmallProf` writes its output into a file called *smallprof.out*, by default. The format of the file looks like this:

```
<num> <time> <ctime> <line>:<text>
```

When the program has terminated, the output may be examined and sorted using any standard text filtering utilities. Something like the following may be sufficient:

```
$> cat smallprof.out | grep \d*: | sort -k3 | tac | head -n20
```

```
251215 1.65674 7.68000 75: if ( $word =~ /($regex)/ ) {
251215 0.03264 4.40000 79: debug("word: $i_wd ".$has ?
'matches' :
251215 0.02693 4.10000 81: return $has;
260643 0.02841 4.07000 128: if ( $debug ) {
260643 0.02601 4.04000 126: my $message = shift;
251215 0.02641 3.91000 73: my $has = 0;
251215 0.03311 3.71000 70: my $i_wd = shift;
251215 0.02699 3.69000 72: my $regex = shift;
251215 0.02766 3.68000 71: my $word = shift;
50243 0.59726 1.00000 59: $count{$i_LINES}{cons} =
50243 0.48175 0.92000 61: $count{$i_LINES}{spec} =
50243 0.00644 0.89000 56: my $i_cons = matches($i_word, $word,
50243 0.48837 0.88000 63: $count{$i_LINES}{caps} =
50243 0.00516 0.88000 58: my $i_caps = matches($i_word, $word,
'^[(A-
50243 0.00631 0.81000 54: my $i_spec = matches($i_word, $word,
'^[a-
50243 0.00496 0.80000 57: my $i_vows = matches($i_word, $word,
50243 0.00688 0.80000 53: $i_word++;
50243 0.48469 0.79000 62: $count{$i_LINES}{only} =
50243 0.48928 0.77000 60: $count{$i_LINES}{vows} =
50243 0.00683 0.75000 55: my $i_only = matches($i_word, $word,
'^[^a-
```

You can immediately see a slightly different focus to the subroutine profiling modules, and we start to see exactly which line of code is taking the most time. That regex line is looking a bit suspicious, for example. Remember that these tools are supposed to be used together, there is no single best way to profile your code, you need to use the best tools for the job.

See also `Apache::SmallProf` which hooks `Devel::SmallProf` into `mod_perl`.

## Devel::FastProf

`Devel::FastProf` is another Perl line profiler. This was written with a view to getting a faster line profiler, than is possible with for example `Devel::SmallProf`, because it's written in C. To use `Devel::FastProf`, supply the `-d` argument to Perl:

```
$> perl -d:FastProf wordmatch -f perl5db.pl
```

```
<...multiple lines snipped...>
```

```
wordmatch report for perl5db.pl:
lines in file: 9428
words in file: 50243
words with special (non-word) characters: 20480
words with only special (non-word) characters: 7790
words with only consonants: 4801
words with only capital letters: 1316
words with only vowels: 1701
```

`Devel::FastProf` writes statistics to the file `fastprof.out` in the current directory. The output file, which can be specified, can be interpreted by using the `fprofpp` command-line program.

```
$> fprofpp | head -n20
```

```
# fprofpp output format is:
# filename:line time count: source
wordmatch:75 3.93338 251215: if ( $word =~ /($regex)/ ) {
wordmatch:79 1.77774 251215: debug("word: $i_wd ".$( $has ? 'matches' :
'does not match')." chars: /$regex/");
wordmatch:81 1.47604 251215: return $has;
wordmatch:126 1.43441 260643: my $message = shift;
wordmatch:128 1.42156 260643: if ( $debug ) {
wordmatch:70 1.36824 251215: my $i_wd = shift;
wordmatch:71 1.36739 251215: my $word = shift;
wordmatch:72 1.35939 251215: my $regex = shift;
```

Straightaway we can see that the number of times each line has been called is identical to the `Devel::SmallProf` output, and the sequence is only very slightly different based on the ordering of the amount of time each line took to execute, `if ( $debug ) { and my $message = shift;`, for example. The differences in the actual times recorded might be in the algorithm used internally, or it could be due to system resource limitations or contention.

See also the `DBIx::Profiler` which will profile database queries running under the `DBIx::*` namespace.

## Devel::NYTProf

`Devel::NYTProf` is the **next generation** of Perl code profiler, fixing many shortcomings in other tools and implementing many cool features. First of all it can be used as either a *line* profiler, a *block* or a *subroutine* profiler, all at once. It can also use sub-microsecond (100ns) resolution on systems which provide `clock_gettime()`. It can be started and stopped even by the program being profiled.

It's a one-line entry to profile `mod_perl` applications. It's written in `c` and is probably the fastest profiler available for Perl. The list of coolness just goes on. Enough of that, let's see how to it works - just use the familiar `-d` switch to plug it in and run the code.

```
$> perl -d:NYTProf wordmatch -f perl5db.pl
```

```
wordmatch report for perl5db.pl:
lines in file: 9427
words in file: 50243
words with special (non-word) characters: 20480
words with only special (non-word) characters: 7790
words with only consonants: 4801
words with only capital letters: 1316
words with only vowels: 1701
```

NYTProf will generate a report database into the file `nytprof.out` by default. Human readable reports can be generated from here by using the supplied `nytprofhtml` (HTML output) and `nytprofcsv` (CSV output) programs. We've used the unix sytem `html2text` utility to convert the `nytprof/index.html` file for convenience here.

```
$> html2text nytprof/index.html
```

```
Performance Profile Index
For wordmatch
Run on Fri Sep 26 13:46:39 2008
Reported on Fri Sep 26 13:47:23 2008
```

```
Top 15 Subroutines -- ordered by exclusive time
```

Calls	P	F	Inclusive Time	Exclusive Time	Subroutine
251215	5	1	13.09263	10.47692	main:: matches
260642	2	1	2.71199	2.71199	main:: debug
1	1	1	0.21404	0.21404	main:: report
2	2	2	0.00511	0.00511	XSLoader:: load (xsub)
14	14	7	0.00304	0.00298	Exporter:: import
3	1	1	0.00265	0.00254	Exporter:: as_heavy
10	10	4	0.00140	0.00140	vars:: import
13	13	1	0.00129	0.00109	constant:: import
1	1	1	0.00360	0.00096	FileHandle:: import
3	3	3	0.00086	0.00074	warnings::register:: import
9	3	1	0.00036	0.00036	strict:: bits
13	13	13	0.00032	0.00029	strict:: import
2	2	2	0.00020	0.00020	warnings:: import
2	1	1	0.00020	0.00020	Getopt::Long:: ParseOptionSpec
7	7	6	0.00043	0.00020	strict:: unimport

For more information see the full list of 189 subroutines.

The first part of the report already shows the critical information regarding which subroutines are using the most time. The next gives some statistics about the source files profiled.

```
Source Code Files -- ordered by exclusive time then name
```

Stmts	Exclusive Time	Avg.	Reports	Source File

	2699761	15.66654	6e-06	line	.	block	.	sub wordmatch
	35	0.02187	0.00062	line	.	block	.	sub IO/Handle.pm
	274	0.01525	0.00006	line	.	block	.	sub Getopt/Long.pm
	20	0.00585	0.00029	line	.	block	.	sub Fcntl.pm
	128	0.00340	0.00003	line	.	block	.	
sub	Exporter/Heavy.pm							
	42	0.00332	0.00008	line	.	block	.	sub IO/File.pm
	261	0.00308	0.00001	line	.	block	.	sub Exporter.pm
	323	0.00248	8e-06	line	.	block	.	sub constant.pm
	12	0.00246	0.00021	line	.	block	.	
sub	File/Spec/Unix.pm							
	191	0.00240	0.00001	line	.	block	.	sub vars.pm
	77	0.00201	0.00003	line	.	block	.	sub FileHandle.pm
	12	0.00198	0.00016	line	.	block	.	sub Carp.pm
	14	0.00175	0.00013	line	.	block	.	sub Symbol.pm
	15	0.00130	0.00009	line	.	block	.	sub IO.pm
	22	0.00120	0.00005	line	.	block	.	sub IO/Seekable.pm
	198	0.00085	4e-06	line	.	block	.	
sub	warnings/register.pm							
	114	0.00080	7e-06	line	.	block	.	sub strict.pm
	47	0.00068	0.00001	line	.	block	.	sub warnings.pm
	27	0.00054	0.00002	line	.	block	.	sub overload.pm
	9	0.00047	0.00005	line	.	block	.	sub SelectSaver.pm
	13	0.00045	0.00003	line	.	block	.	sub File/Spec.pm
	2701595	15.73869		Total				
	128647	0.74946		Average				
		0.00201	0.00003	Median				
		0.00121	0.00003	Deviation				

Report produced by the NYTProf 2.03 Perl profiler, developed by Tim Bunce and Adam Kaplan.

At this point, if you're using the *html* report, you can click through the various links to bore down into each subroutine and each line of code. Because we're using the text reporting here, and there's a whole directory full of reports built for each source file, we'll just display a part of the corresponding

wordmatch-line.html file, sufficient to give an idea of the sort of output you can expect from this cool tool.

```
$> html2text nytprof/wordmatch-line.html
```

```
Performance Profile -- -block view--line view--sub view-
For wordmatch
Run on Fri Sep 26 13:46:39 2008
Reported on Fri Sep 26 13:47:22 2008
```

File wordmatch

```
Subroutines -- ordered by exclusive time
|Calls|P|F|Inclusive|Exclusive|Subroutine|
|-----|-----|-----|-----|-----|-----|
|251215|5|1|13.09263|10.47692|main::matches|
|260642|2|1|2.71199|2.71199|main::debug|
|1|1|1|0.21404|0.21404|main::report|
|0|0|0|0|0|main::BEGIN|

|Line|Stmts.|Exclusive|Avg.  |Code|
|-----|-----|-----|-----|-----|
|1|1|0.00000|0.00000|#!/usr/bin/perl|
|2|1|0.00000|0.00000|use strict;|
|3|3|0.00086|0.00029|# spent 0.00003s making 1 calls to
strict::|
|4|3|0.01563|0.00521|# spent 0.00012s making 1 calls to
warnings::|
|5|1|0.00000|0.00000|import|
|6|1|0.00000|0.00000|use warnings;|
|7|1|0.00000|0.00000|=head1 NAME|
|8|1|0.00000|0.00000|filewords - word analysis of input file
<...snip...>
|62|1|0.00445|0.00445|print report( %count );|
|63|1|0.21404|0.21404|# spent 0.21404s making 1 calls to
main::report|
```



| 112 | | | | |

Oodles of very useful information in there - this seems to be the way forward.

See also `Devel::NYTProf::Apache` which hooks `Devel::NYTProf` into `mod_perl`.

## SORTING

Perl modules are not the only tools a performance analyst has at their disposal, system tools like `time` should not be overlooked as the next example shows, where we take a quick look at sorting. Many books, theses and articles, have been written about efficient sorting algorithms, and this is not the place to repeat such work, there's several good sorting modules which deserve taking a look at too: `Sort::Maker`, `Sort::Key` spring to mind. However, it's still possible to make some observations on certain Perl specific interpretations on issues relating to sorting data sets and give an example or two with regard to how sorting large data volumes can effect performance. Firstly, an often overlooked point when sorting large amounts of data, one can attempt to reduce the data set to be dealt with and in many cases `grep()` can be quite useful as a simple filter:

```
@data = sort grep { /$filter/ } @incoming
```

A command such as this can vastly reduce the volume of material to actually sort through in the first place, and should not be too lightly disregarded purely on the basis of it's simplicity. The `KISS` principle is too often overlooked - the next example uses the simple system `time` utility to demonstrate. Let's take a look at an actual example of sorting the contents of a large file, an apache logfile would do. This one has over a quarter of a million lines, is 50M in size, and a snippet of it looks like this:

# logfile

```
188.209-65-87.adsl-dyn.isp.belgacom.be - - [08/Feb/2007:12:57:16 +0000]
"GET /favicon.ico HTTP/1.1" 404 209 "-" "Mozilla/4.0 (compatible; MSIE
6.0; Windows NT 5.1; SV1)"
188.209-65-87.adsl-dyn.isp.belgacom.be - - [08/Feb/2007:12:57:16 +0000]
"GET /favicon.ico HTTP/1.1" 404 209 "-" "Mozilla/4.0 (compatible; MSIE
6.0; Windows NT 5.1; SV1)"
151.56.71.198 - - [08/Feb/2007:12:57:41 +0000] "GET /suse-on-vaio.html
HTTP/1.1" 200 2858 "http://www.linux-on-laptops.com/sony.html" "Mozilla/5.0
(Windows; U; Windows NT 5.2; en-US; rv:1.8.1.1) Gecko/20061204
Firefox/2.0.0.1"
151.56.71.198 - - [08/Feb/2007:12:57:42 +0000] "GET /data/css HTTP/1.1"
404 206 "http://www.rfi.net/suse-on-vaio.html" "Mozilla/5.0 (Windows; U;
Windows NT 5.2; en-US; rv:1.8.1.1) Gecko/20061204 Firefox/2.0.0.1"
151.56.71.198 - - [08/Feb/2007:12:57:43 +0000] "GET /favicon.ico
HTTP/1.1" 404 209 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.2; en-US;
rv:1.8.1.1) Gecko/20061204 Firefox/2.0.0.1"
217.113.68.60 - - [08/Feb/2007:13:02:15 +0000] "GET / HTTP/1.1" 304 -
-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)"
217.113.68.60 - - [08/Feb/2007:13:02:16 +0000] "GET /data/css HTTP/1.1"
404 206 "http://www.rfi.net/" "Mozilla/4.0 (compatible; MSIE 6.0; Windows
NT 5.1; SV1)"
deborato.isac.cnr.it - - [08/Feb/2007:13:03:58 +0000] "GET
/suse-on-vaio.html HTTP/1.1" 200 2858
"http://www.linux-on-laptops.com/sony.html" "Mozilla/5.0 (compatible;
Konqueror/3.4; Linux) KHTML/3.4.0 (like Gecko)"
deborato.isac.cnr.it - - [08/Feb/2007:13:03:58 +0000] "GET /data/css
HTTP/1.1" 404 206 "http://www.rfi.net/suse-on-vaio.html" "Mozilla/5.0
```

```
(compatible; Konqueror/3.4; Linux) KHTML/3.4.0 (like Gecko)"
  debora.to.isac.cnr.it - - [08/Feb/2007:13:03:58 +0000] "GET
/favicon.ico HTTP/1.1" 404 209 "-" "Mozilla/5.0 (compatible; Konqueror/3.4;
Linux) KHTML/3.4.0 (like Gecko)"
  195.24.196.99 - - [08/Feb/2007:13:26:48 +0000] "GET / HTTP/1.0" 200
3309 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1.8.0.9)
Gecko/20061206 Firefox/1.5.0.9"
  195.24.196.99 - - [08/Feb/2007:13:26:58 +0000] "GET /data/css HTTP/1.0"
404 206 "http://www.rfi.net/" "Mozilla/5.0 (Windows; U; Windows NT 5.1;
fr; rv:1.8.0.9) Gecko/20061206 Firefox/1.5.0.9"
  195.24.196.99 - - [08/Feb/2007:13:26:59 +0000] "GET /favicon.ico
HTTP/1.0" 404 209 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.1; fr;
rv:1.8.0.9) Gecko/20061206 Firefox/1.5.0.9"
  crawl1.cosmixcorp.com - - [08/Feb/2007:13:27:57 +0000] "GET /robots.txt
HTTP/1.0" 200 179 "-" "voyager/1.0"
  crawl1.cosmixcorp.com - - [08/Feb/2007:13:28:25 +0000] "GET /links.html
HTTP/1.0" 200 3413 "-" "voyager/1.0"
  fhm226.internetdsl.tpnet.pl - - [08/Feb/2007:13:37:32 +0000] "GET
/suse-on-vaio.html HTTP/1.1" 200 2858
"http://www.linux-on-laptops.com/sony.html" "Mozilla/4.0 (compatible; MSIE
6.0; Windows NT 5.1; SV1)"
  fhm226.internetdsl.tpnet.pl - - [08/Feb/2007:13:37:34 +0000] "GET
/data/css HTTP/1.1" 404 206 "http://www.rfi.net/suse-on-vaio.html"
"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)"
  80.247.140.134 - - [08/Feb/2007:13:57:35 +0000] "GET / HTTP/1.1" 200
3309 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR
1.1.4322)"
  80.247.140.134 - - [08/Feb/2007:13:57:37 +0000] "GET /data/css
HTTP/1.1" 404 206 "http://www.rfi.net" "Mozilla/4.0 (compatible; MSIE 6.0;
Windows NT 5.1; .NET CLR 1.1.4322)"
  pop.compuscan.co.za - - [08/Feb/2007:14:10:43 +0000] "GET / HTTP/1.1"
200 3309 "-" "www.clamav.net"
  livebot-207-46-98-57.search.live.com - - [08/Feb/2007:14:12:04 +0000]
"GET /robots.txt HTTP/1.0" 200 179 "-" "msnbot/1.0
(+http://search.msn.com/msnbot.htm)"
  livebot-207-46-98-57.search.live.com - - [08/Feb/2007:14:12:04 +0000]
"GET /html/oracle.html HTTP/1.0" 404 214 "-" "msnbot/1.0
(+http://search.msn.com/msnbot.htm)"
  dslb-088-064-005-154.pools.arcor-ip.net - - [08/Feb/2007:14:12:15
+0000] "GET / HTTP/1.1" 200 3309 "-" "www.clamav.net"
  196.201.92.41 - - [08/Feb/2007:14:15:01 +0000] "GET / HTTP/1.1" 200
3309 "-" "MOT-L7/08.B7.DCR MIB/2.2.1 Profile/MIDP-2.0
Configuration/CLDC-1.1"
```

The specific task here is to sort the 286,525 lines of this file by Response Code, Query, Browser, Referring Url, and lastly Date. One solution might be to use the following code, which iterates over the files given on the command-line.

```
# sort-apache-log

#!/usr/bin/perl -n

use strict;
use warnings;

my @data;
```

```

LINE:
while ( <> ) {
    my $line = $_;
    if (
        $line =~ m/^(
            ([\w\.\-]+)           # client
            \s*-\s*-\s*\[
            ([^]]+)              # date
            \]\s*"w+\s*
            (\S+)                # query
            [^"]+\s*
            (\d+)                # status
            \s+\S+\s+"[^"]*" \s+"
            ([^"]*)"             # browser
            "
            .*
        )$/x
    ) {
        my @chunks = split(/ +/, $line);
        my $ip      = $1;
        my $date    = $2;
        my $query   = $3;
        my $status  = $4;
        my $browser = $5;

        push(@data, [$ip, $date, $query, $status, $browser, $line]);
    }
}

my @sorted = sort {
    $a->[3] cmp $b->[3]
    ||
    $a->[2] cmp $b->[2]
    ||
    $a->[0] cmp $b->[0]
    ||
    $a->[1] cmp $b->[1]
    ||
    $a->[4] cmp $b->[4]
} @data;

foreach my $data ( @sorted ) {
    print $data->[5];
}

exit 0;

```

When running this program, redirect `STDOUT` so it is possible to check the output is correct from following test runs and use the system `time` utility to check the overall runtime.

```

$> time ./sort-apache-log logfile > out-sort

real    0m17.371s
user    0m15.757s

```

sys 0m0.592s

The program took just over 17 wallclock seconds to run. Note the different values `time` outputs, it's important to always use the same one, and to not confuse what each one means.

#### Elapsed Real Time

The overall, or wallclock, time between when `time` was called, and when it terminates. The elapsed time includes both user and system times, and time spent waiting for other users and processes on the system. Inevitably, this is the most approximate of the measurements given.

#### User CPU Time

The user time is the amount of time the entire process spent on behalf of the user on this system executing this program.

#### System CPU Time

The system time is the amount of time the kernel itself spent executing routines, or system calls, on behalf of this process user.

Running this same process as a `Schwarzian Transform` it is possible to eliminate the input and output arrays for storing all the data, and work on the input directly as it arrives too. Otherwise, the code looks fairly similar:

#### # sort-apache-log-schwarzian

```
#!/usr/bin/perl -n

use strict;
use warnings;

print

    map $_->[0] =>

    sort {
        $a->[4] cmp $b->[4]
        ||
        $a->[3] cmp $b->[3]
        ||
        $a->[1] cmp $b->[1]
        ||
        $a->[2] cmp $b->[2]
        ||
        $a->[5] cmp $b->[5]
    }
    map [ $_, m/^(
        ([\w\.\-]+)           # client
        \s*\-\s*\-\s*\[
        ([^]]+)              # date
        \]\s*" \w+ \s*
        (\S+)                 # query
        [^"]+\s*
        (\d+)                 # status
        \s+\S+\s+" [^"]*" \s+"
        ([^"]*)              # browser
    ]
```

```

        . *
    )$/xo ]

    => <>;

    exit 0;

```

Run the new code against the same logfile, as above, to check the new time.

```

$> time ./sort-apache-log-schwarzian logfile > out-schwarz

real    0m9.664s
user    0m8.873s
sys     0m0.704s

```

The time has been cut in half, which is a respectable speed improvement by any standard. Naturally, it is important to check the output is consistent with the first program run, this is where the unix system `cksum` utility comes in.

```

$> cksum out-sort out-schwarz
3044173777 52029194 out-sort
3044173777 52029194 out-schwarz

```

BTW. Beware too of pressure from managers who see you speed a program up by 50% of the runtime once, only to get a request one month later to do the same again (true story) - you'll just have to point out your only human, even if you are a Perl programmer, and you'll see what you can do...

## LOGGING

An essential part of any good development process is appropriate error handling with appropriately informative messages, however there exists a school of thought which suggests that log files should be *chatty*, as if the chain of unbroken output somehow ensures the survival of the program. If speed is in any way an issue, this approach is wrong.

A common sight is code which looks something like this:

```

    logger->debug( "A logging message via process-id: $$ INC: " .
Dumper(\%INC) )

```

The problem is that this code will always be parsed and executed, even when the debug level set in the logging configuration file is zero. Once the `debug()` subroutine has been entered, and the internal `$debug` variable confirmed to be zero, for example, the message which has been sent in will be discarded and the program will continue. In the example given though, the `\%INC` hash will already have been dumped, and the message string constructed, all of which work could be bypassed by a debug variable at the statement level, like this:

```

    logger->debug( "A logging message via process-id: $$ INC: " .
Dumper(\%INC) ) if $DEBUG;

```

This effect can be demonstrated by setting up a test script with both forms, including a `debug()` subroutine to emulate typical `logger()` functionality.

```

# ifdebug

#!/usr/bin/perl

```

```

use strict;
use warnings;

use Benchmark;
use Data::Dumper;
my $DEBUG = 0;

sub debug {
    my $msg = shift;

    if ( $DEBUG ) {
        print "DEBUG: $msg\n";
    }
};

timethese(100000, {
    'debug' => sub {
        debug( "A $0 logging message via process-id: $$" .
Dumper(\%INC) )
    },
    'ifdebug' => sub {
        debug( "A $0 logging message via process-id: $$" .
Dumper(\%INC) ) if $DEBUG
    },
});

```

Let's see what Benchmark makes of this:

```

$> perl ifdebug
Benchmark: timing 100000 iterations of constant, sub...
ifdebug:  0 wallclock secs ( 0.01 usr +  0.00 sys =  0.01 CPU) @
10000000.00/s (n=100000)
(warning: too few iterations for a reliable count)
debug: 14 wallclock secs (13.18 usr +  0.04 sys = 13.22 CPU) @
7564.30/s (n=100000)

```

In the one case the code, which does exactly the same thing as far as outputting any debugging information is concerned, in other words nothing, takes 14 seconds, and in the other case the code takes one hundredth of a second. Looks fairly definitive. Use a `$DEBUG` variable BEFORE you call the subroutine, rather than relying on the smart functionality inside it.

### Logging if DEBUG (constant)

It's possible to take the previous idea a little further, by using a compile time `DEBUG` constant.

```

# ifdebug-constant

#!/usr/bin/perl

use strict;
use warnings;

use Benchmark;
use Data::Dumper;
use constant

```

```

        DEBUG => 0
    ;

    sub debug {
        if ( DEBUG ) {
            my $msg = shift;
            print "DEBUG: $msg\n";
        }
    };

    timethese(100000, {
        'debug' => sub {
            debug( "A $0 logging message via process-id: $$" .
Dumper(\%INC) )
        },
        'constant' => sub {
            debug( "A $0 logging message via process-id: $$" .
Dumper(\%INC) ) if DEBUG
        },
    });

```

Running this program produces the following output:

```

$> perl ifdebug-constant
Benchmark: timing 100000 iterations of constant, sub...
   constant:  0 wallclock secs (-0.00 usr +  0.00 sys = -0.00 CPU) @
-7205759403792793600000.00/s (n=100000)
      (warning: too few iterations for a reliable count)
   sub: 14 wallclock secs (13.09 usr +  0.00 sys = 13.09 CPU) @
7639.42/s (n=100000)

```

The `DEBUG` constant wipes the floor with even the `$debug` variable, clocking in at minus zero seconds, and generates a "warning: too few iterations for a reliable count" message into the bargain. To see what is really going on, and why we had too few iterations when we thought we asked for 100000, we can use the very useful `B::Deparse` to inspect the new code:

```

$> perl -MO=Deparse ifdebug-constant

use Benchmark;
use Data::Dumper;
use constant ('DEBUG', 0);
sub debug {
    use warnings;
    use strict 'refs';
    0;
}
use warnings;
use strict 'refs';
timethese(100000, {'sub', sub {
    debug "A $0 logging message via process-id: $$" . Dumper(\%INC);
}
, 'constant', sub {
    0;
}
});

```

```
ifdebug-constant syntax OK
```

The output shows the `constant()` subroutine we're testing being replaced with the value of the `DEBUG` constant: zero. The line to be tested has been completely optimized away, and you can't get much more efficient than that.

## POSTSCRIPT

This document has provided several way to go about identifying hot-spots, and checking whether any modifications have improved the runtime of the code.

As a final thought, remember that it's not (at the time of writing) possible to produce a useful program which will run in zero or negative time and this basic principle can be written as: *useful programs are slow* by their very definition. It is of course possible to write a nearly instantaneous program, but it's not going to do very much, here's a very efficient one:

```
$> perl -e 0
```

Optimizing that any further is a job for `p5p`.

## SEE ALSO

Further reading can be found using the modules and links below.

## PERLDOCS

For example: `perldoc -f sort`.

*perfaq4*.

*perlfork*, *perlfunc*, *perlretut*, *perltlthrtut*.

*threads*.

## MAN PAGES

*time*.

## MODULES

It's not possible to individually showcase all the performance related code for Perl here, naturally, but here's a short list of modules from the CPAN which deserve further attention.

```
Apache::DProf
Apache::SmallProf
Benchmark
DBIx::Profiler
Devel::AutoProfiler
Devel::DProf
Devel::DProfLB
Devel::FastProf
Devel::GraphVizProf
Devel::NYTProf
Devel::NYTProf::Apache
Devel::Profiler
Devel::Profile
Devel::Profit
Devel::SmallProf
Devel::WxProf
POE::Devel::Profiler
Sort::Key
Sort::Maker
```

## URLS

Very useful online reference material:

[http://www.ccl4.org/~nick/P/Fast\\_Enough/](http://www.ccl4.org/~nick/P/Fast_Enough/)

<http://www-128.ibm.com/developerworks/library/l-optperl.html>

<http://perlbuzz.com/2007/11/bind-output-variables-in-dbi-for-speed-and-safety.html>

[http://en.wikipedia.org/wiki/Performance\\_analysis](http://en.wikipedia.org/wiki/Performance_analysis)

<http://apache.perl.org/docs/1.0/guide/performance.html>

<http://perlgolf.sourceforge.net/>

[http://www.sysarch.com/Perl/sort\\_paper.html](http://www.sysarch.com/Perl/sort_paper.html)

[http://www.unix.org.ua/oreilly/perl/prog/ch08\\_03.htm](http://www.unix.org.ua/oreilly/perl/prog/ch08_03.htm)

## AUTHOR

Richard Foley <[richard.foley@rfi.net](mailto:richard.foley@rfi.net)> Copyright (c) 2008