Profilers identify bottlenecks in Perl programs

RUN-TIME SPEED

Dominic Ziegler, Fotolia

Instead of souped up CPUs, simple programming tricks are often all it takes to speed up a program. Profilers can discover bottlenecks that need more TLC from the developer. **BY MICHAEL SCHILLI**

If you don't spend all day playing video games, you will be quite happy without the latest and greatest CPU, and you can save money on hardware while saving power and reducing your personal CO_2 footprint.

THE AUTHOR

Michael Schilli works as a software engineer with Yahoo! in Sunnyvale, California. He is the author of *Goto Perl 5* (German) and *Perl Power* (English), both published by Addison-Wesley, and he can be contacted at *mschilli@perlmeister.com*. Michael's homepage is at *http://perlmeister.com*.

But what happens if a Perl program does not perform as well as you would like? As a first step, you might be able to accelerate the program with very little effort by optimizing the code at neuralgic points.

In most cases, just one or two targeted improvements will help you achieve 90 percent of the total optimization potential. The remaining 10 percent are more likely to involve wide-ranging, architectural changes that take 10 times as long to implement and make the resulting program so difficult to maintain that more experienced developers are likely to turn down the option.

Profilers are programs that discover bottlenecks, or hotspots. They show you where the program spends most of its time and help to optimize these crucial sections in a targeted way.

Perl has many tools of this kind. For example, if a module uses the Log4perl logging framework, it is simple to display the number of milliseconds (msec) that elapse along with the log messages, thus creating a timescale.

Finding Bottlenecks

The simple test script *amzntest* (Listing 1) uses the CPAN *Net::Amazon* module to retrieve book details from the Amazon Web API (assuming you enter a valid token, which you can get from Amazon Web Services [2]).

A request takes about a second, but what does the script use this time for – picking up the data or the analyzing the XML response? Adding

```
use Log::Log4perl;
Log::Log4perl->init("timer.l4p");
```

```
mschilli@mybox:~/DEV/articles/nytprof/eg
$ cmdcolor l4ptimer.out
$ ./amzn
2008/10/04 10:37:37 78 request: params = $VAR1 =
2008/10/04 10:37:37 79 urlstr=http://webservices
2008/10/04 10:37:37 79 Fetching http://webservic
2008/10/04 10:37:38 739 Received [ <?xml version=
2008/10/04 10:37:38 787 xmlref_add (after):$VAR1
2008/10/04 10:37:38 788 Received valid XML (1 ite
2008/10/04 10:37:38 789 Creating new Book Propert
2008/10/04 10:37:38 790 Setting AverageRating via
2008/10/04 10:37:38 790 Setting TotalReviews via
2008/10/04 10:37:38 790 Setting TotalReviewPages
2008/10/04 10:37:38 791 Adding review A must for
2008/10/04 10:37:38 791 Adding review embedded de
2008/10/04 10:37:38 791 Adding review The very BE
2008/10/04 10:37:38 791 Adding review A 'must hav
2008/10/04 10:37:38 791 Adding review Best Linux
[1]  Sreekrishnan Venkateswaran, "Essential Linux Device
     Drivers (Prentice Hall Open Source Software
     Development Series)", 2008, $35.99, 0132396556
$
```

Figure 1: Log4perl outputs the time of day and the number of milliseconds (79msec) that have elapsed since the program was launched.

```
mschilli@mybox:~/DEV/articles/nytprof/eg
$ ./amzncache
2008/10/04 11:36:15 134 request: params = $VAR1 =
2008/10/04 11:36:15 134 urlstr=http://webservices
2008/10/04 11:36:15 134 Fetching http://webservic
2008/10/04 11:36:15 144 Serving from cache
2008/10/04 11:36:15 144 Received [ <?xml version=
2008/10/04 11:36:15 176 xmlref_add (after):$VAR1
2008/10/04 11:36:15 177 Received valid XML (1 ite
2008/10/04 11:36:15 178 Creating new Book Propert
2008/10/04 11:36:15 179 Setting AverageRating via
2008/10/04 11:36:15 179 Setting TotalReviews via
2008/10/04 11:36:15 179 Setting TotalReviewPages
2008/10/04 11:36:15 179 Adding review A must for
2008/10/04 11:36:15 179 Adding review embedded de
2008/10/04 11:36:15 180 Adding review The very BE
2008/10/04 11:36:15 180 Adding review A 'must hav
2008/10/04 11:36:15 180 Adding review Best Linux
[1]  Sreekrishnan Venkateswaran, "Essential Linux Device
     Drivers (Prentice Hall Open Source Software
     Development Series)", 2008, $35.99, 0132396556
$
```

Figure 2: The use of a web cache removes the need to issue a web request in case of repetitions, thus reducing the run time by 75 percent.

to the script header enables the Log4perl statements embedded in *Net::Amazon* by loading the *timer.l4p* configuration file. This outputs the date (*%d*) and the number of milliseconds since the program started (*%r*) for each message it logs. The logging level is set to *DEBUG* and is thus fairly verbose.

To cut down on lengthy log messages, the configuration file in Listing 2 uses the abbreviation *%S* to define a so-called Custom Cspec. This appears in the layout, instead of the more typical *%m* (for the log message).

The associated Perl subroutine returns a shorter, 25-character message to the screen appender. The layout closes with a platform-dependent newline character defined by *%n*.

Listing 1: amzntest

```
01 #!/usr/bin/perl
02 use warnings;
03 use strict;
04 use Net::Amazon;
05 use Net::Amazon::Request::ASIN;
06
07 my $asin = "0132396556";
08
09 my $ua = Net::Amazon->new(
10     token => 'XXXXXXXXXXXXXXX',
11 );
12
13 my $req = Net::Amazon::Request::ASI
   N->new(
14     asin  => $asin,
15 );
16
17 my $resp = $ua->request($req);
18
19 if($resp->is_success()) {
20     print $resp->as_string(), "\n";
21 } else {
22     print "Error: ",
23           $resp->message(), "\n";
24 }
```

Quick Fix

Figure 1 shows that *Net::Amazon* sends a web request to Amazon 79msec after the program launches; however, the XML parser does not start working until 739msec have elapsed. Thus, the web request takes 75 percent of the 800msec required by the script.

As you will agree, it makes sense to use a web cache for frequently retrieved ASIN numbers, and *Net::Amazon* enables this if we ask it to (see Listing 3). This really does reduce the script run time, which drops to 180msec for repetitions (see Figure 2) with the use of *File::Cache* as a persistent cache.

Quick Results

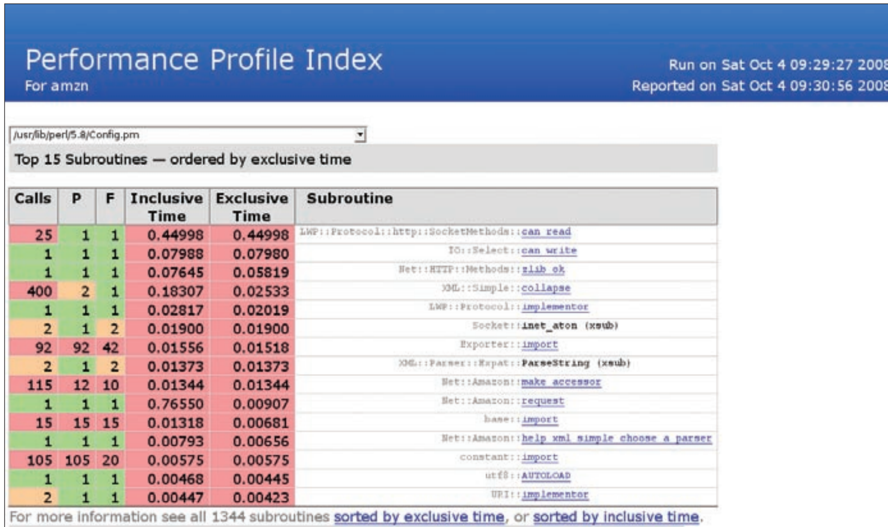
Now you might argue that tricks of this kind aren't always going to help, but the decisive issue here is that five lines of code and a minute spent thinking about

```
smallprof.out (~/DEV/articles/nytprof/eg) - VIM
        ================ SmallProf version 2.02 ================
     Profile of /home/mschilli/PERL/lib/perl5/site_perl/5.8.5/LWP/ Page 420
   =================================================================
 count wall tm  cpu time line
     0   0.00000   0.00000   393:}
     0   0.00000   0.00000   394:
     0   0.00000   0.00000   395:sub can_read {
    34   0.00000   0.00000   396:    my($self, $timeout) = @_;
    34   0.00000   0.00000   397:    my $fbits = '';
    34   0.00007   0.00000   398:    vec($fbits, fileno($self), 1) = 1;
    34   0.37738   0.00000   399:    my $nfound = select($fbits, undef, undef,
    34   0.00001   0.00000   400:    die "select failed: $!" unless defined
    34   0.00006   0.00000   401:    return $nfound > 0;
     0   0.00000   0.00000   402:}
     0   0.00000   0.00000   403:
                                                          17094,1       66%
```

Figure 3: The 25,000 lines of output from SmallProf reveals that waiting for the Amazon server takes longest.

Listing 2: timer.l4p

```
1 log4perl.logger          = DEBUG, App
2 log4perl.appender.App     = Log::Log4perl::Appender::Screen
3 log4perl.appender.App.layout = PatternLayout
4 log4perl.appender.App.layout.ConversionPattern = %d %r %S%n
5 log4perl.PatternLayout.cspec.S = sub { substr($_[1], 0, 25) }
```



Performance Profile Index
For amzn
Run on Sat Oct 4 09:29:27 2008
Reported on Sat Oct 4 09:30:56 2008

/usr/lib/perl/5.8/Config.pm

Top 15 Subroutines — ordered by exclusive time

Calls	P	F	Inclusive Time	Exclusive Time	Subroutine
25	1	1	0.44998	0.44998	LWP::Protocol::http::SocketMethods::can_read
1	1	1	0.07988	0.07980	IO::Select::can_write
1	1	1	0.07645	0.05819	Net::HTTP::Methods::zlib_ok
400	2	1	0.18307	0.02533	XML::Simple::collapse
1	1	1	0.02817	0.02019	LWP::Protocol::implementor
2	1	2	0.01900	0.01900	Socket::inet_aton (xsub)
92	92	42	0.01556	0.01518	Exporter::import
2	1	2	0.01373	0.01373	XML::Parser::Expat::ParseString (xsub)
115	12	10	0.01344	0.01344	Net::Amazon::make_accessor
1	1	1	0.76550	0.00907	Net::Amazon::request
15	15	15	0.01318	0.00681	base::import
1	1	1	0.00793	0.00656	Net::Amazon::help_xml_simple_choose_a_parser
105	105	20	0.00575	0.00575	constant::import
1	1	1	0.00468	0.00445	utf8::AUTOLOAD
2	1	1	0.00447	0.00423	URI::implementor

For more information see all 1344 subroutines sorted by exclusive time, or sorted by inclusive time.

Figure 4: The 15 functions that waste the most time.

options achieved a speed boost of 400 percent.

SmallProf Line Profiler

Of course, some modules do not have very sophisticated logging mechanisms. Thankfully, Perl has a number of profilers that measure the amount of time each line of source code uses.

Table 1 shows the major CPAN profilers for various tasks in ascending release order.

The CPAN *Devel::SmallProf* line profiler autonomously measures run times and formats the data for analysis. If you launch the script you want to test via *perl -d:SmallProf ./amzn*, the profiler creates a *smallprof.out* file that lists the time used by each line in each module used.

In the case of a fairly complex action like a web request and ensuing XML analysis of the returned data, the resultant file can be enormous – in this case, it weighed in at no fewer than 25,792 lines.

Discovering the worst time hogs is not an easy task, but the shell command

```
sort -k 2,2 smallprof.out | less
```

sorts the file numerically on the second field from the left. This field contains the wall clock time in seconds used by a line of source code, which is the time actually used whether the task gave the CPU something to do or just left it idle, waiting for external events, such as incoming network packages, for example. In addition, the third field contains the CPU seconds, which is the computational time actually used.

If you scroll down to line 17,104 in *smallprof.out*, as shown in Figure 3, you will see that the line responsible for the wait issues a *select* command. The *can_read* function in the *LWP::Protocol::http::SocketMethods* module is responsible for this.

Table 1: Perl Profilers

Name	Year
Subroutine-Level Profilers	
Devel::DProf	1995
Devel::AutoProfiler	2002
Devel::Profiler	2002
Devel::Profile	2003
Devel::DProfLB	2006
Devel::WxProf	2008
Statement-Level Profilers	
Devel::SmallProf	1997
Devel::FastProf	2005
Devel::NYTProf	2008
Devel::Profit	2008

NYTProf

The CPAN module *Devel::NYTProf* is a relatively new development [3]. The strange-sounding name came about because the module was developed for the New York Times and based on the *Devel::FastProf* codebase. Their IT department decided to release the source code.

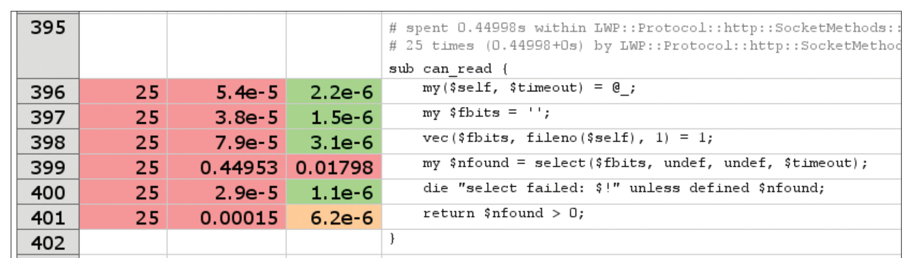
This excellent profiler is maintained by DBI inventor Tim Bunce [4] today. Bunce presented his work to the Perl community at OSCON 2008 [5]. Although I was in the audience, I couldn't concentrate on his talk because I was up next with my Log4perl talk.

After the CPAN shell-assisted installation (*perl -MCPAN -e'install Devel::NYTProf'*), you type *perl -d:NYTProf amzn* to launch the profiler on the script. To convert its binary logfile *nytprof.out* to professionally formatted HTML, use the *nytprofhtml* script included with the package.

Firing up a browser and pointing it to the *index.html* file in the newly created *nytprof* directory (URL *file:///...nytprof/index.html*) reveals what has been going on. The table in Figure 4 lists the hotspots. It shows the number of calls, the number of places (P) they have been issued, the files (F) they are located in, and the time spent executing each function. The tool differentiates between "exclusive time" and "inclusive time," the former referring to the time spent exclu-

Listing 3: Web Cache

```
01 use Cache::File;
02 my $cache = Cache::File->new(
03    cache_root     => '/tmp/mycache',
04    default_expires  => '30 min',
05 );
06
07 my $ua = Net::Amazon->new(
08     token =>
   '0AQN5ZBDW6NXV9M60N82',
09     cache => $cache,
10 );
```



395				# spent 0.44998s within LWP::Protocol::http::SocketMethods:: # 25 times (0.44998+0s) by LWP::Protocol::http::SocketMethod sub can_read {
396	25	5.4e-5	2.2e-6	my($self, $timeout) = @_;
397	25	3.8e-5	1.5e-6	my $fbits = '';
398	25	7.9e-5	3.1e-6	vec($fbits, fileno($self), 1) = 1;
399	25	0.44953	0.01798	my $nfound = select($fbits, undef, undef, $timeout);
400	25	2.9e-5	1.1e-6	die "select failed: $!" unless defined $nfound;
401	25	0.00015	6.2e-6	return $nfound > 0;
402				}

Figure 5: The select command in the LWP code waits for a response from Amazon and is thus responsible for a major part of the program run time.

Source Code Files — ordered by exclusive time then name

Stmts	Exclusive Time	Avg.	Reports	Source File
574	0.45488	0.00079	line • block • sub	LWP/Protocol/http.pm
58	0.08136	0.00140	line • block • sub	IO/Select.pm
9963	0.03505	3.5e-6	line • block • sub	XML/Simple.pm
1003	0.02351	2.3e-5	line • block • sub	Net/Amazon.pm
107	0.02229	0.00021	line • block • sub	IO/Socket/INET.pm
2770	0.01483	5.4e-6	line • block • sub	Log/Log4perl/Logger.pm
1697	0.01349	8.0e-6	line • block • sub	Exporter.pm
49	0.01214	0.00025	line • block • sub	Data/Dumper.pm
1082	0.01031	9.5e-6	line • block • sub	Exporter/Heavy.pm
246	0.01020	4.1e-5	line • block • sub	utf8_heavy.pl
306	0.01009	3.3e-5	line • block • sub	LWP/UserAgent.pm
5315	0.00959	1.8e-6	line • block • sub	XML/Parser/Style/Tree.pm
101	0.00924	9.2e-5	line • block • sub	XML/Parser/Expat.pm
122	0.00875	7.2e-5	line • block • sub	IO/Compress/Base/Common.pm
1380	0.00782	5.7e-6	line • block • sub	Net/HTTP/Methods.pm
68	0.00762	0.00011	line • block • sub	IO/Uncompress/Base.pm
60	0.00750	0.00013	line • block • sub	IO/Compress/Base.pm
296	0.00690	2.3e-5	line • block • sub	HTTP/Message.pm
1803	0.00597	3.3e-6	line • block • sub	constant.pm
848	0.00587	6.9e-6	line • block • sub	Log/Log4perl.pm
82	0.00587	7.2e-5	line • block • sub	Compress/Raw/Zlib.pm

Figure 6: The NYTProf shows how much time the code spends in each module and how many statements were processed.

sively in the function's code and the latter referring to the total time, including the subfunctions called by the function.

Single-Click Analysis

The HTML links the function names with HTML pages providing details on the function code – a very practical idea that lets the user flip back and forth and greatly facilitates the analysis. Figure 5, for example, shows details that were very difficult to obtain with SmallProf; that is, the *select()* command in the *can_read()* function in an LWP module is causing the delays. On an open network socket, it waits for the first signs of the response returned by Amazon.

NYTProf supports three different reporting modes for displaying the time spent by the source code: line mode (one time value per line), block mode (one time value per block of Perl), and sub mode (one time value per function).

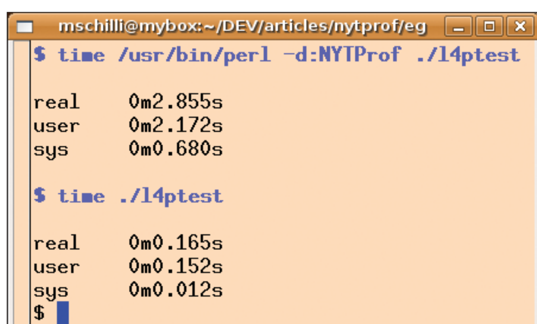


Figure 7: NYTProf can slow down fast programs – considerably in some cases.

Clicking toggles the performance detective through the different display variants.

The module-level view of the profiler in Figure 6 also reveals some interesting facts. The XML analysis of the Amazon web response took 9,963 lines of code and 400 calls to the *collapse()* function *XML::Simple*.

Because it ran at full speed, the whole kit and caboodle took just 35msec, but this just goes to show how complex it is to parse XML.

Not Without Side Effects

Just like any other profiler, *Devel::NYTProf* also involves some overhead that can completely falsify the measured results in some cases. The profiler's activities are particularly invasive if a program is not waiting for external events, such as network traffic or disk access, which are magnitudes of scale slower.

For a program running at full CPU speed, the run time can be 10 times longer if you enable the profiler.

Figure 7 shows the effect the NYT profiler has on the short *l4ptest* test program (Listing 4). The program configures Log4perl for the *$INFO* logging priority and then issues *$DEBUG* messages, which should be suppressed because

Listing 4: l4ptest

```
#!/usr/local/bin/perl -w
use strict;

use Log::Log4perl qw(:easy);
Log::Log4perl->easy_init($INFO);

for(1..100_000) {
    DEBUG "waah!";
}
```

of their lower priority. Log4perl has optimized this case; after all, a disabled logging system shouldn't have any measurable effect.

The script running without the profiler achieves around 100,000 calls in about 100msec; the script run time is about 10 times longer with NYTProf enabled.

This by no means detracts from the quality of the profiler, but it is important to take this into consideration when measuring script run time and interpreting the results.

The module also is quite at home in the Apache server's *mod_perl*. Adding *PerlModule Devel::NYTProf::Apache* to the configuration loads the module and tells it to append profile data to the */tmp/nytprof.$$.out* file for incoming requests, where *$$* represents the PID for the Apache process handling the request.

Calling *nytprofhtml* again creates a collection of web pages for in-depth analysis of the web application's performance that could very well point you to critical code sections that need improvement. ■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/resources/article_code*

[2] Amazon Web Services (developer token application): *http://amazon.com/soap*

[3] Devel::NYTProf: *http://search.cpan.org/dist/Devel-NYTProf/*

[4] Tim Bunce blog "NYTProf v2 – the background story," *http://blog.timbunce.org/2008/07/16/nytprof-v2-the-background-story/*

[5] Tim Bunce at OSCON 2008: *http://en.oreilly.com/oscon2008/public/schedule/speaker/6816*