Understanding Load Averages and Stretch Factors

Neil J. Gunther

Abstract

No doubt you have often seen and even made use of those three little numbers called "load averages" that appear in shell commands like `procinfo` *and* `uptime`*. But how well do you understand them? Why are there always three numbers? How are they calculated? And what is their origin? As well as answering these questions in the following, the concept of* **stretch factor** *is introduced as a way to enhance load average data for improved performance management of symmetric multiprocessor and multicore servers.*

1 Introduction

Most Linux system administrators are familiar with those three little numbers that appear in shell commands like `procinfo`, `uptime`, `top`, and remote host `ruptime`. For example, `uptime` emits:

```
9:40am up 9 days, load average: 0.02, 0.01, 0.00
```

Note that the load average metrics are not available in their own right. Rather, they are always included within the output of other commands. While the load average is a well-known to Linux system administrators, its meaning is often rather poorly understood.

To start with, even the word "load" means different things to different people. To a system administrator it tends to imply the number of *active users* on a server, whereas to a performance analyst it tends to imply the *utilization* of the server. The man page for the `uptime` states:

```
uptime gives a one line display of ...  the
system load averages for the past 1, 5, and 15
minutes.
```

which does explain why there are three numbers, but does not explain what the word *load* means in this context. The man page for `procinfo` states:

```
The average number of jobs running, followed
by the number of runnable processes and ...
```

For somewhat deeper explanations, we can turn to experts [1] who warn us:

> *The load average tries to measure the number of active processes at any time. As a measure of CPU utilization, the load average is simplistic, poorly defined, but far from useless.*

That's encouraging (Not!). As you will find out, the load average is a "poor" measure of processor utilization precisely because it is *not* a measure of CPU utilization.

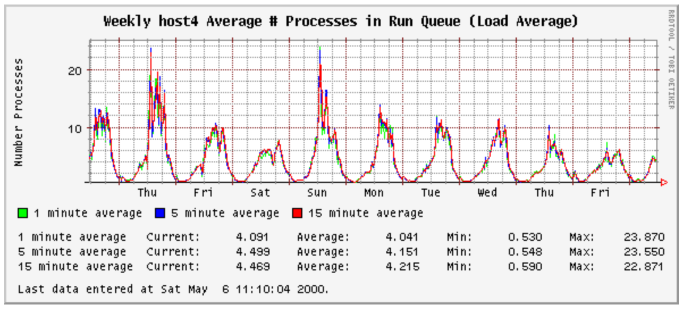
So, *load* is somehow related to active processes, but what is an "average" load? Our experts [1] sometimes answer a question with a question:

> *What's high? As usual, that depends on your system. Ideally, you'd like a load average under, say, 3, ... Ultimately, "high" means high enough so that you don't need uptime to tell you that the system is overloaded.*

Huh? Why a load average less than 3? They continue:

> *...different systems will behave differently under the same load average. . . .running a single CPU-bound background job...can bring response to a crawl even though the load average remains quite low.*

Somewhat surprisingly, we shall find out in the next section that this last statement is absolutely correct!



```
1 minute average   Current:    4.091    Average:    4.041    Min:    0.530    Max:    23.870
5 minute average   Current:    4.499    Average:    4.151    Min:    0.548    Max:    23.550
15 minute average  Current:    4.469    Average:    4.215    Min:    0.590    Max:    22.871
Last data entered at Sat May  6 11:10:04 2000.
```

Figure 1: Periodically sampled load averages presented as a time series during a 1-week window. The 1-, 5-, and 15-minute values appear as superimposed green, blue, and red curves

Modern graphical tools, like Orca [2], can provide a much broader perspective on load averages by representing them as a time series (Fig. 1). Blair Zajac, author of Orca points out:

> *If long term trends indicate increasing figures, more or faster CPUs will eventually be necessary unless load can be displaced. For ideal utilization of your CPU, the maximum value here should be equal to the number of CPUs in the box.*

This statement certainly recognizes that the run-queue might be serviced by multiple processors. Unfortunately, it also suggests that any form of queueing (i.e., waiting) is a bad thing. As we shall see in Section 5, nothing could be further from the truth. No waiting might be desirable for a number-crunching application (see Sect. 5.2), but for commercial workloads some amount of queueing is both expected and desirable (see Sect. 5.1). Afterall, that's the reason Linux has a run-queue in the first place.

We are getting nowhere fast this way. To clarify things further, I performed a set of controlled experiments on a single-processor Linux box.

2 Controlled Experiments

Experimental load averages were sampled over a one-hour period (3600 seconds) on an otherwise quiescent single-CPU Linux box. The tests consisted of two phases:

1. Two CPU-intensive jobs were initiated as background processes and allowed to execute for 2,100 seconds.
2. These two processes were stopped simultaneously but load average measurements were continued for another 1,500 seconds.

The following Perl script was used to sample the load average every 5 seconds using the `uptime` command.

```
#! /usr/bin/perl -w
$sample_interval = 5; # seconds

# Fire up background cpu-intensive tasks ...
system("./burncpu &");
system("./burncpu &");

# Perpetually monitor the load average via uptime
# and emit it as tab-separated fields for possible
# use in a spreadsheet program.
while (1) {
    @uptime = split (/ /, `uptime`);
    foreach $up (@uptime) {
        # collect the timestamp
        if ($up =~ m/(\d\d:\d\d:\d\d)/) {
            print "$1\t";
        }
        # collect the three load metrics
        if ($up =~ m/(\d{1,}\.\d\d)/) {
            print "$1\t";
        }
    }
    print "\n";
    sleep ($sample_interval);
}
```

A C program called `burncpu.c` was designed to waste CPU cycles. Output from `top` shows the two instances of `burncpu` ranked as highest CPU consumers during measurement period when `getload` was running.

```
  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
20048 neil      25   0   256  256   212 R    30.6  0.0   0:32   0 burncpu
20046 neil      25   0   256  256   212 R    29.3  0.0   0:32   0 burncpu
15709 mir       24   0  9656 9656  4168 R    25.6  1.8  45:32   0 kscience.kss
 1248 root      15   0 66092  10M  1024 S     9.5  2.1 368:25   0 X
20057 neil      16   0  1068 1068   808 R     2.3  0.2   0:01   0 top
 1567 mir       15   0 39228  38M 14260 S     1.3  7.6  40:10   0 mozilla-bin
 1408 mir       15   0   340  296   216 S     0.7  0.0  50:33   0 autorun
 1397 mir       15   0  2800 1548   960 S     0.1  0.3   1:57   0 kdeinit
20044 neil      15   0  1516 1516  1284 S     0.1  0.2   0:00   0 perl
    1 root      15   0   156  128   100 S     0.0  0.0   0:04   0 init
```

Figure 2 shows that the 1-minute load average reaches a value of 2.0 after 300 s into the test, the 5-minute load average reaches 2.0 around 1,200 seconds, while the 15-minute load average would reach 2.0 at approximately 4,500 seconds but the processes were *killed* at 2100 seconds.

Readers with a background in electrical engineering will immediately spot the striking resemblance between the data in Figure 2 and the voltage curves produced by charging and discharging an RC-circuit. We shall learn why this analogy is more than mere whimsy in Section 4. Notice that the maximum load during the test is equivalent to the number of CPU-intensive processes running at the time of the measurements. The "fins" in the top curve are a result of various demons waking up temporarily and then going back to sleep. They can be considered noise in the experimental setup. By the way, if there was just a single process running, the load average would not be greater than 1.0 and you might be forgiven for drawing the wrong conclusion that load average *is* a direct measure of CPU utilization. I say this because I made this mistake in one experiment.

My next objective was to explain why the load average data from these experiments exhibit the characteristics seen in Figure 2. For that, I needed to explore the Linux kernel

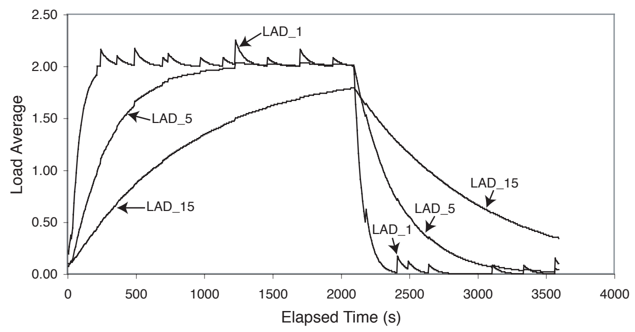


Figure 2: Load average data (LAD) collected on a controlled Linux platform during a 1 hour period. LAD_1, LAD_5, LAD_15 signify the 1-, 5-, and 15-minute metrics, respectively

code that calculates the load average metrics. I chose to use the Linux 2.6.20.1 source code available online at `http://lxr.linux.no/source/`, complete with cross-referencing hyperlinks, for easier navigation and enhanced readability.

3 Kernel Code

Looking at the source code for the CPU scheduler, we find the following C function called `calc_load` in `http://lxr.linux.no/source/kernel/timer.c`.

```
unsigned long avenrun[3];

EXPORT_SYMBOL(avenrun);

/*
 * calc_load - given tick count, update the avenrun load estimates.
 * This is called while holding a write_lock on xtime_lock.
 */
static inline void calc_load(unsigned long ticks)
{
        unsigned long active_tasks; /* fixed-point */
        static int count = LOAD_FREQ;

        count -= ticks;
        if (unlikely(count < 0)) {
                active_tasks = count_active_tasks();
                do {
                    CALC_LOAD(avenrun[0], EXP_1, active_tasks);
                    CALC_LOAD(avenrun[1], EXP_5, active_tasks);
                    CALC_LOAD(avenrun[2], EXP_15, active_tasks);
                    count += LOAD_FREQ;
                } while (count < 0);
        }
}
```

This is the primary routine that calculates the load average metrics. Essentially, it checks to see if the sample period has expired, resets the sampling counter, and calls the subroutine `CALC_LOAD` to calculate each of the 1-minute, 5-minute, and 15-minute metrics respectively. The sampling interval used for `LOAD_FREQ` is `5*HZ`. How long is that interval?

Every Linux platform has a clock implemented in hardware. This hardware clock has a constant ticking rate by which everything else in the system is synchronized. To make this ticking rate known to the system, it sends an interrupt to the kernel on every clock *tick*. The actual interval between ticks differs depending on the type of platform, e.g., most Linux systems have the CPU tick interval set to 10 ms of wall-clock time.

The specific definition of the tick rate is contained in a constant labeled `HZ` that is maintained in a system-specific header file called `param.h`. For the online Linux source code we are using here, you can see the value is 100 for an Intel platform in `lxr.linux.no/source/include/asm-i386/param.h`, and for a SPARC-based system in `lxr.linux.no/source/include/asm-sparc/param.h`. However, it is defined differently for a MIPS processor in `lxr.linux.no/source/include/asm-mips/param.h`. The statement:

```
#define       HZ    100
```

in the header file means that one second of wall-clock time is divided into 100 ticks. In other words, we could say that a clock interrupt occurs with a *frequency* of once every 100th of a second, or 1 tick = 1 s/100 or 10 milliseconds. Conversely, the C macro at line 73:

```
#define CT_TO_SECS(x)   ((x) / HZ)
```

is used to convert the number of ticks to seconds.

The constant labeled `HZ` should be read as the frequency *divisor* and not literally as the SI unit of frequency *cycles per second*, the latter actually having the symbol *Hz*. Thus, `5 * HZ` means five times the value of the constant called `HZ`. Furthermore, since `HZ` is equivalent to 100 ticks, 5×100 ticks = 500 ticks, it follows that 500 ticks is the same as 500×10 milliseconds or an interval of 5 s.

So, `CALC_LOAD` is called once every 5 s, and not 5 times per second as some people mistakenly think. Also, be careful not to confuse this sampling period of 5 s with the *reporting* periods of 1, 5, and 15 minutes.

The C macro `CALC_LOAD` does the real work of calculating the load average and it is defined in `http://lxr.linux.no/source/include/linux/sched.h` of the following code fragment:

```
98 /*
99  * These are the constant used to fake the fixed-point load-average
100  * counting. Some notes:
101  *  - 11 bit fractions expand to 22 bits by the multiplies: this gives
102  *    a load-average precision of 10 bits integer + 11 bits fractional
103  *  - if you want to count load-averages more often, you need more
104  *    precision, or rounding will get you. With 2-second counting freq,
105  *    the EXP_n values would be 1981, 2034 and 2043 if still using only
106  *    11 bit fractions.
107  */
108 extern unsigned long avenrun[];   /* Load averages */
109
110 #define FSHIFT       11          /* nr of bits of precision */
111 #define FIXED_1      (1<<FSHIFT)  /* 1.0 as fixed-point */
112 #define LOAD_FREQ    (5*HZ)       /* 5 sec intervals */
113 #define EXP_1        1884         /* 1/exp(5sec/1min) as fixed-point */
114 #define EXP_5        2014         /* 1/exp(5sec/5min) */
115 #define EXP_15       2037         /* 1/exp(5sec/15min) */
116
117 #define CALC_LOAD(load,exp,n) \
118         load *= exp; \
119         load += n*(FIXED_1-exp); \
120         load >>= FSHIFT;
```

Several questions immediately come to mind when reading this code:

1. Where do those strange numbers 1884, 2014, 2037 come from?
2. What role do they play in calculating the load averages?
3. What does the `CALC_LOAD` code actually do?

I will address these questions in the next section. First, we need to make a brief detour into the world of fixed-point arithmetic.

The cryptic comment that precedes the `CALC_LOAD` macro alerts us to the fact that *fixed-point*, rather than floating-point, arithmetic is used to calculate the load average. Since the calculations are done in the kernel, the presumption is that fixed-point arithmetic is more efficient, although no explicit justification is given.

A fixed-point representation means that only a fixed number of digits, either decimal or binary, are used to express any number, including those that have a fractional part (mantissa) following the decimal point. Suppose, for example, that 4 bits of precision were allowed in the mantissa. Then, numbers like:

$$0.1234,\ -12.3401,\ 1.2000,\ 1234.0001$$

can be represented exactly. On the other hand, numbers like:

$$0.12346,\ -8.34051$$

cannot be represented exactly and would generally be rounded to:

$$0.1235,\ -8.3405$$

As comment warns, too much successive rounding can cause otherwise insignificant errors to become compounded into significant errors. One way around this problem is to increase the number of bits used to express the mantissa, assuming enough storage is available to accommodate the greater precision.

←10 bits→		⟵					11 bits					⟶
0000000001	.	0	0	0	0	0	0	0	0	0	0	0
1		0	0	0	0	0	0	0	0	0	0	0
Bit position:		10	9	8	7	6	5	4	3	2	1	0

Table 1: Bit positions for digits in 10.11 fixed-point format

The comment also indicates that there are 10 bits allowed for the integer part of the number and 11 bits for the fractional part. This is called an $M.N = 10.11$ format. The rules of fixed-point addition are the same as for integers. The important difference occurs with fixed-point multiplication. The product of multiplying two $M.N$ fixed-point numbers is:

$$M.N \times M.N = (M + M).(N + N) \tag{1}$$

To get back to $M.N$ format, the lower-order bits are eliminated by shifting N bits.

There are several fixed-point constants used in the `CALC_LOAD` routine. The first of these is the number '1' itself, which is labeled `FIXED_1` (see line 111). In Table 1, the top row corresponds to `FIXED_1` expressed in 10.11 format, whereas the leading zeros and the decimal point have been dropped from second row. The digits of the resulting binary 100000000000_2 are indexed 0 through 11 on the last row of Table 1. This establishes that `FIXED_1` is equivalent to 2^{11} or 2048_{10} in decimal notation.

Using the C left-shift bitwise operator (<<), 100000000000_2 can be written as 1 << 11 in agreement with line 111 of the macro code. Alternatively, we can write FIXED_1 as a decimal integer:

$$\texttt{FIXED\_1} = 2048_{10}\,, \tag{2}$$

to simplify calculation of the remaining constants: EXP_1, EXP_5, and EXP_15, for the 1-, 5-, and 15 minute metrics, respectively.

Consider the 1-minute metric as an example. If we denote the sample period as σ and the reporting period as τ, then:

$$\texttt{EXP\_1} \equiv \mathrm{e}^{-\sigma/\tau}\,. \tag{3}$$

I have already established that $\sigma = 5$ seconds and for the 1-minute metric, $\tau = 60$ second. Furthermore, the decimal value of EXP_1 is:

$$\mathrm{e}^{-5/60} = 0.92004441463\,. \tag{4}$$

To convert eqn.(4) to a 10.11 fixed-point fraction, we only need to multiply it by the fixed-point constant FIXED_1 or '1' to produce:

$$\lfloor 2048 \times 0.92004441463 \rfloor = 1884_{10}\,, \tag{5}$$

and round it to the nearest 11-bit integer.

Each of the other magic numbers can be calculated in the same way, and the results are summarized in Table 2. The

Table 2: Default magic numbers for 5 second sampling

Base	Sec.	1. exp($-5/\tau$)	Rounded	Binary
τ_1	60	1884.25	1884_{10}	11101011100_2
τ_5	300	2014.15	2014_{10}	11111011110_2
τ_{15}	900	2036.65	2037_{10}	11111110101_2

results in Table 2 agree with the kernel defines:

```
#define EXP_1     1884
#define EXP_5     2014
#define EXP_15    2037
```

If the sampling rate was decreased to a 2 second interval, the constants would need to be changed to those in the fourth column of Table 3. This explains how the three

Table 3: Magic numbers for 2 second sampling

Base	Sec.	1. exp($-2/\tau$)	Rounded	Binary
τ_1	60	1980.86	1981_{10}	11110111101_2
τ_5	300	2034.39	2034_{10}	11111110010_2
τ_{15}	900	2043.45	2043_{10}	11111111011_2

magic numbers arise. Next, we need to understand what the CALC_LOAD function actually does with them.

4 Load Average Revealed

Mathematically, the CALC_LOAD (see line 117) is equivalent to taking the current value of the variable load and multiplying it by a factor called exp. This value of load is then added to a term comprising the number of active processes n multiplied by another variable called FIXED_1-exp. The last line of the macro decimalizes the value of load.

We also know that the macro variable exp is equivalent to $\mathrm{e}^{-\sigma/\tau}$ by virtue of equation (3) and FIXED_1-exp is equivalent to $1 - \mathrm{e}^{-\sigma/\tau}$ by virtue of equations (2) and (3). Writing CALC_LOAD macro in more conventional mathematical notation produces:

$$L(t) = L(t-1)\ \mathrm{e}^{-\sigma/\tau} + n(t)\ (1 - \mathrm{e}^{-\sigma/\tau}) \tag{6}$$

where $L(t)$ is the current value of the load variable, $L(t-1)$ is its value from the previous sample, and $n(t)$ is number of currently active Linux processes.

A Linux process can be in one of about half a dozen states (depending on how you count) of which running, runnable (R in the ps command), and sleeping (S in ps) are the three primary states. A nice animation of these states and the possible transitions between them can be found at http://www.linux-tutorial.info/modules.php?name=Tutorial&pageid=84. Each load-average metric is based on the total number of processes that are:

1. *runnable* and waiting in the scheduler run-queue
2. currently *running* or exectuting on a processor

This accounts for the cryptic remark about "active processes" made by our experts in Section 1. In queueing theory terminology, the total active processes is called a *queue* [3]. It literally means, not just those processes that are in the waiting line (the so-called run-queue), but also those that are currently being serviced (i.e., running).

So, CALC_LOAD is the fixed-point arithmetic version of equation (6). How equation (6) behaves is best understood by examining some special cases.

4.1 Empty Run-Queue

First, consider the case where the process run-queue is empty, i.e., $n(t) = 0$. Recall from Section 1 that the run-queue includes not just those Linux processes that are waiting in the run-queue (*runnable*), but also those currently executing (*running*) on CPUs.

Setting $n(t) = 0$ in eqn.(6) produces:

$$L(t) = L(t-1)\,\mathrm{e}^{-\sigma/\tau} \tag{7}$$

If we iterate eqn.(7) between $t = t_0$ and $t = T$ we get:

$$L(T) = L(t_0)\,\mathrm{e}^{-\sigma t/\tau} \tag{8}$$

A plot of equation (8) is shown in Figure 3 for the three load average reporting metrics (τ). It corresponds to time-dependent exponential *decay*.

In other words, it is this load average calculation that corresponds to the fall-off in the observed load between $t_0 = 2,100$ and $T = 3,600$ in Figure 3, which compares the data from Figure 2 (labeled *LAD*) with the curve produced by equation (8) labeled *EMA*.

4.2 Full Run-Queue

The second special case corresponds to Section 2 with the run-queue is consistently occupied by two processes. The second term in equation (6) now dominates, and iterating from time $t = t_0$ to time $t = T$ produces:

$$L(T) = 2\,L(t_0)\,(1 - \mathrm{e}^{-\sigma t/\tau}) \tag{9}$$

Similarly, a plot of equation (9) is shown in Figure 4 for the three load average reporting metrics (τ) tracks the data (labeled *LAD*) as a monotonically *increasing* functions.

We see that equation (9) is responsible for the observed *rise* in load between $t_0 = 0$ and $T = 2,100$ in Fig. 2.

Table 4: Calculated rise times for the data in Fig. 4

Load avg. parameter	Time constant	Estimated rise time (s)
τ_1	60	300
τ_5	300	1500
τ_{15}	900	4500

Having already noted earlier that the curves in Figure 2 resemble the voltage characteristic of an RC-circuit, we can take that analogy a step further. In circuit theory, it is known that the rise time is approximately 5 times the characteristic time constant τ. In CALC_LOAD, $\tau_1 = 60$ seconds,

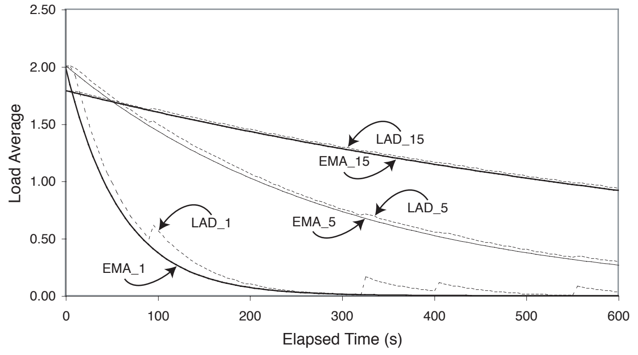


Figure 3: Comparison of load average data (*LAD*) with equation (8) (*EMA*) for 600 seconds after process termination in Figure 2

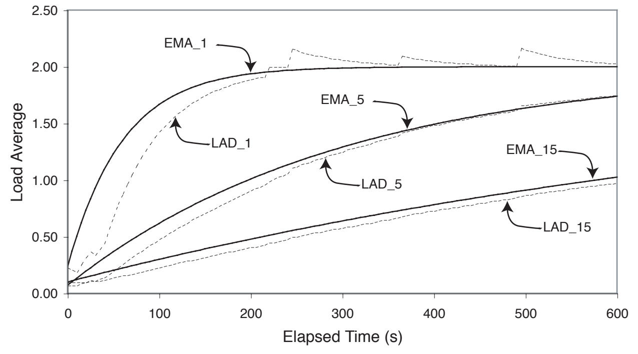


Figure 4: Comparison of load average data (*LAD*) with equation (9) (*EMA*) during the measurement period $0 \leq T \leq 600$ seconds in Figure 2

therefore the rise time can be estimated as $5\ \tau_1 \approx 300$ seconds; exactly as observed in Figure 4. The other rise times are summarized in Table 4.

It turns out that there is nothing particularly novel about the way the load average is calculated. In fact, a common technique for processing highly variable raw data for subsequent analysis, is to apply some kind of smoothing function to that data. The general relationship between the raw input data and the smoothed output data is given by:

$$\underbrace{Y(t)}_{\text{smoothed}} = Y(t-1) + \underbrace{\alpha}_{\text{constant}} \left[\underbrace{X(t)}_{\text{raw}} - Y(t-1) \right] \tag{10}$$

The smoothing function in equation (10) is an *exponential filter* or *exponentially-smoothed moving average* (EMA) of the type used in financial forecasting (see e.g., http://bigcharts.marketwatch.com) and signal processing. The parameter α in equation (10) is commonly called the *smoothing constant*, while $(1 - \alpha)$ is called the *damping factor*. Moreover, both these factors can be directly related to the corresponding factors in equation (6) [3]. The magnitude of the smoothing coefficient $(0 \leq \alpha \leq 1)$ determines how much the current forecast must be corrected for error in the previous iteration of the forecast.

Table 5: Damping factors for CALC_LOAD

Timebase parameter	Damping factor $\mathrm{e}^{-\sigma/\tau}$	Smoothing constant α
τ_1	0.9200	0.0800 ($\approx$ 8%)
τ_5	0.9835	0.0165 ($\approx$ 2%)
τ_{15}	0.9945	0.0055 ($\approx$ 1%)

Notice that the exponential damping factor for τ_1 in Table 5 agrees with the value in equation (4) to four decimal places. The 1-minute load average metric has the least damping, or about 8% correction, because it is the most responsive to instantaneous changes in the length of the run-queue. Conversely, the 15-minute load average has the most damping, or only 1% correction, because it is the least responsive metric.

5 Stretch Factors

Inevitably, the question arises: What is a good load average? A simple question; there should be a simple answer, right? The expert comment in Section 1 about an ideal load average of "3" notwithstanding, one could legitimately wonder if a *rule-of-thumb* could be constructed for quantitatively assessing the load average on multicore and multiprocessor platforms? Moreover, since we now know that the load average is an exponentially damped moving average of activity in the process run-queue, we could convert the question to, How long should my queue be?

As with all seemingly benign performance questions, they just *look* simple. In fact, the question about ideal load average values is ill-posed. Unfortunately, it is like asking, How long is a piece of string? A slightly cynical answer is, not so long that you strangle yourself with unintended consequences. Similarly with queues. Queues should not be so long that they introduce unintended consequences. What might those consequences be?

Long queues correspond to long response times, so it's really the response time metric that should get your attention. One consequence is that a long queue cause "poor response times", but that depends on what *poor* means. In most performance management tools, there is a disconnect between the measured run-queue length and the user-perceived response times. Another problem is that queue length is an **absolute** measure, whereas what is really needed is a *relative* performance measure. Even the words, *poor* and *good* are relative terms. Such a relative metric is called the *stretch factor* [4], and it measures the mean queue length relative to the mean number of requests already in service. It is expressed in multiples of *service units*. A stretch factor of one, means no waiting time is involved.

What makes the stretch factor really useful is that it can easily be compared with service level targets. Service targets are usually expressed in certain types of business units, e.g., *quotes per hour* is a common business unit of work for an insurance industry. The expected service level is called the *service level objective* or SLO, and is expressed as multiples of the relevant service unit. An SLO might be documented as:

> *The average user response time is not to exceed 15 service units between the peak operating hours of 10 am and 2 pm.*

Which is the same as saying the SLO shall not exceed a stretch factor of 15.

Table 6: Stretch factor definitions

m	number of processors or cores
Q	measured load average
ρ	measured processor utilization

Using the symbols defined in Table 6, the stretch factor f can be calculated as the ratio:

$$f = \frac{Q}{m\rho} \tag{11}$$

From the controlled experiments of Section 2, we know that $m = 1$ because it was a single-process box, $Q = 2$ for the 1-minute load average, and $\rho = 1$ because the workload was CPU-bound. Substituting these values into equation (11) gives a stretch factor of $f = 2$. This result tells us that the expected time for any process to complete execution is two service periods. Notice that we don't have to know what is the *actual* service period. The stretch factor and the load average happen to be identical in this case, because the processes are running on a single processor and they are CPU-intensive.

Let's look at a couple of real-world examples to see how this stretch factor concept can be applied.

5.1 Anti-spam Farm

All major email hosting services run spam analyzers. A typical configuration might consist of a set of specialized servers, each raking over email text using a filtering tool like *SpamAssassin* (`http://spamassassin.apache.org/`). One such well-known, and therefore heavily trafficed web portal, has a battery of some 100 servers, each comprising 2 dual-cores, all performing 7 by 24 email scanning. Typical daily spam-filtering statistics are shown in Table 7.

Table 7: Daily Spam Server Statistics

Number of CPUs	4
Spam detected	33901
Ham accepted	23123
Emails processed	57024
Emails per hour	2376
Per CPU/hour	594
CPU busy%	99
Secs per email	6
Load average	97.36

A load balancer was used to distribute work into the server farm. The effectiveness of the load balancer was monitored using 1-minute load averages. The sample of these load averages from 50 of the servers is shown in Figure 5 reveals an imbalance of work in the farm.

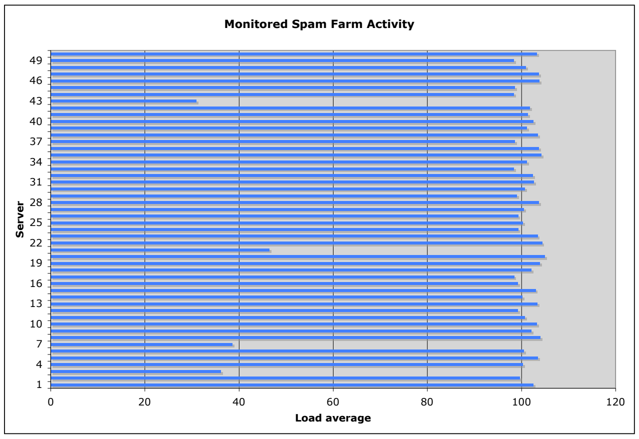


Figure 5: Measured load averages showing the unbalanced work distribution across a sample of 50 spam-farm servers

Some system administration questions include:

1. Why is there a load imbalance?
2. Are most servers overdriven as a consequence of the load imbalance?
3. Is a load average of $Q = 97.36$ emails desirable?
4. What should be the actual server performance?
5. How many additional servers will be needed in the next fiscal year to maintain current scanning performance at higher loads?

Let's substitute the load average into equation (11) to calculate the stretch factor:

$$f = \frac{97.36}{4 \times 0.99} = 24.59 \tag{12}$$

From Table 7, the average time to scan an email message (S) is 6 seconds. So, a stretch factor of $f = 25$ service periods implies that it takes about $25 \times 6 = 150$ seconds or 2.5 minutes from the time an email message reaches the portal until it lands in the intended user's email box.

An absolute value of $Q = 97.36$ for the load average tells us very little. The relative stretch factor, however, tells us how many service periods the spam filtering is costing. The answer to question 3, about desirablity, then depends on the agreed upon service targets. At least now, such questions can be addressed quantitatively instead of speculatively.

PDQ in Python

PDQ (*Pretty Damn Quick*) is a modeling tool for analyzing the performance characteristics of computational resources e.g., processors disks, and a set of processes that make requests for those resources. A PDQ model is analyzed using algorithms based on queueing theory. The current release facilitates building and analyzing performance models in C, Perl, python, Java and PHP.

The python PDQ functions and procedures used in this section are:

- `pdq.Init()` initializes internal PDQ variables.
- `pdq.CreateOpen()` creates a workload.
- `pdq.CreateNode()` creates a server.
- `pdq.SetDemand()` sets the workload service time on the server resource.
- `pdq.Solve()` calculates performance metrics.
- `pdq.Report()` generates a generic performance report.

More information about the PDQ library can found at: `www.perfdynamics.com/Tools`

- Overview `../PDQ.html`
- Download `../PDQcode.html`
- Manual `../PDQman.html`

PDQ is maintained by the author and Peter Harding.

We can also use the data in Table 7 to answer question 4 with a performance prediction tool like PDQ [3]. Presenting PDQ in detail here would take us too far afield, but the interested reader can see the sidebar *PDQ in Python* and reference [4] to learn more. Here is that spam-server model in PyDQ (PDQ in python):

```
#!/usr/bin/env python
import pdq

# Measured performance parameters
cpusPerServer = 4
emailThruput  = 2376 # emails per hour
scannerTime   = 6.0  # seconds per email

pdq.Init("Spam Farm Model")
# Timebase is SECONDS ...
nstreams = pdq.CreateOpen("Email", float(emailThruput)/3600)
nnodes   = pdq.CreateNode("spamCan", int(cpusPerServer), pdq.MSQ)
pdq.SetDemand("spamCan", "Email", scannerTime)
pdq.Solve(pdq.CANON)
pdq.Report()
```

Running this PDQ model produces a report which contains the following section:

```
               ******   SYSTEM Performance     *******

Metric                     Value    Unit
------                     -----    ----
Workload: "Email"
Number in system        100.7726    Trans
Mean throughput           0.6600    Trans/Sec
Response time           152.6858    Sec
Stretch factor           25.4476
```

The stretch factor predicted by PDQ is a little bigger than we calculated using equation (12). Why is that? To answer this question, we need to look at the section of the PDQ report that presents server performance information.

```
               ******   RESOURCE Performance   *******

Metric          Resource     Work              Value   Unit
------          --------     ----              -----   ----
Throughput      spamCan      Email            0.0660   Trans/Sec
Utilization     spamCan      Email           99.0000   Percent
Queue length    spamCan      Email          100.7726   Trans
Waiting line    spamCan      Email           96.8126   Trans
Waiting time    spamCan      Email          146.6858   Sec
Residence time  spamCan      Email          152.6858   Sec
```

Given the rate at which work is arriving (2376 emails per hour), each CPU should be 99% busy. This utilization is higher than seen in the actual spam farm because of the load imbalance. PDQ is assuming ideal load balance across all servers, so more work is getting done. The predicted load average (*Queue length* metric in the PDQ report) is closer to 100 emails, and therefore, the predicted stretch factor of 25.45 is a little larger than the calculated value of 24.59.

Question 5 asks about the future. Either stretch-factor value was considered to be borderline acceptable under peak load conditions. Since all the servers are close to saturated, one recourse is to upgrade with faster CPUS or, more likely, procure new 4-way servers to handle the expected additional work. PDQ helps to size the number of new servers based on current and expected stretch factors.

Clearly, it is the stretch factor *ratio* that provides a more meaningful indicator for performance management than the absolute load average by itself.

5.2 Number Cruncher

Let's return to the remark in Section 1 made in the context of the Orca tool. We can use a similar PyDQ model to see what it means to have no waiting line with all the CPUs busy. In this case, each Linux process that takes 10 hours to complete because it is transforming oil-exploration data for further analysis by geophysicists. Here is the corresponding PyDQ model:

```
#!/usr/bin/env python
import pdq

processors  = 4     # Same as spam farm example
```

```
arrivalRate = 0.099 # Jobs per hour (very low arrivals)
crunchTime  = 10.0  # Hours (very long service time)

pdq.Init("ORCA LA Model")
s = pdq.CreateOpen("Crunch", arrivalRate)
n = pdq.CreateNode("HPCnode", int(processors), pdq.MSQ)
pdq.SetDemand("HPCnode", "Crunch", crunchTime)
pdq.SetWUnit("Jobs")
pdq.SetTUnit("Hour")
pdq.Solve(pdq.CANON)
pdq.Report()
```

The corresponding PDQ Report contains the following output:

```
                ******   RESOURCE Performance   *******

Metric          Resource     Work              Value   Unit
------          --------     ----              -----   ----
Throughput      HPCnode      Crunch           0.0990   Jobs/Hour
Utilization     HPCnode      Crunch          24.7500   Percent
Queue length    HPCnode      Crunch           0.9965   Jobs
Waiting line    HPCnode      Crunch           0.0065   Jobs
Waiting time    HPCnode      Crunch           0.0656   Hour
Residence time  HPCnode      Crunch          10.0656   Hour
```

The waiting line is essentially zero length and all 4 CPUs are busy, though only 25% utilized. How can this be? If we were to look at the CPU statistics while the system was running, we would observe that each CPU was actually 100% busy. To understand what PDQ is telling us, we need to look at the System Performance section of the PDQ report:

```
                ******   SYSTEM Performance     *******

Metric                     Value    Unit
------                     -----    ----
Workload: "Crunch"
Number in system          0.9965    Jobs
Mean throughput           0.0990    Jobs/Hour
Response time            10.0656    Hour
Stretch factor            1.0066
```

The stretch factor is 1 (service period) because there is no waiting line. On the other hand, it takes 10 hrs for each job to get through the system, so the response time is about 10 hours.

The reason this appears a little odd is due to the fact that PDQ makes predictions based on *steady state* behavior, i.e., how the system looks in the long run [3, 4]. With a service time of 10 hours, we really need to observe the system for much longer than that to see what it looks like in steady state. Much longer here, means on the order of 100 hours or longer. We don't actually need to do that, but PDQ is telling us how things would look if we did. In such a long measurement period, we would not see any new arrivals.

Since the average service period is relatively large, the request rate is correspondingly small, so that no waiting line forms. This, in turn, means that the processor utilization 25% is also low—in in the long view. Looking at the system for just a few minutes while it is crunching 10 hours worth of oil-exploration data, corresponds to an *instantaneous* snapshot of the system, not the steady-state view.

Like the controlled experiments in Section 2, both stretch-factor examples involved CPU-bound workloads to more clearly reveal the relationships between the performance metrics. I/O-bound workloads (either disk or network) will tend to exhibit smaller load averages than CPU-bound work if those processes become suspended or sleep waiting on data (Sect. 4). In that state they are neither runnable nor running and therefore do not contribute to $n(t)$ in equation (6). Conversely, when the Linux I/O driver is performing work, it runs in kernel mode on a CPU and does contribute to $n(t)$.

Returning to the original question about rules-of-thumb for load averages, which opened this section, we see that since Q measures the total number of requests; both waiting and in service. It not a very meaningful quantity because it is an absolute value. Combining it, however, with the number of configured processors (m) and their average measured utilization (ρ), the stretch factor f provides a better performance management metric for symmetric multiprocessor and multicore servers, because it is a relative performance indicator which can be compared directly with established SLOs.

6 Conclusion

The intent of the load average metrics is to provide information about the trend in the growth of the length of the run queue. That's why it reports three metrics. Each tries to capture historical trend information from the run-queue as it was 1-, 5-, and 15-minutes ago.

Compared with today's graphical data display capabilities (Fig. 1), this approach to data representation looks antique. In fact, the load average is one of the earliest forms of operating system instrumentation; the lineage circa 1965 being CTSS → Multics → UNIX → Linux (see \url{www.multicians.org/InstrumentationPaper.html} and [3]). Based on these historical developments, each of the three metrics is an exponentially weighted moving average (EMA) of the sampled run-queue; a well-known algorithm for smoothing data. The load average uses exponential smoothing so that more weight is given to the most recent run-queue samples, thereby avoiding outlier effects, as well as the need to buffer sampled data in the kernel.

It is important to remember that I uncovered these details about the load average calculations by using *controlled measurements* to generate Figure 2. One could never expect to see such "charge-discharge" phenomena by merely staring at a production time-series like Figure 1, no matter how patient you are.

Finally, in Section 5, I presented the stretch factor as a better way to make use of load average data for the performance management of application service level targets on multicore servers.

7 Author

Neil Gunther, M.Sc., Ph.D. is an internationally recognized consultant who founded Performance Dynamics Company in 1994. Prior to that, Dr. Gunther held research and management positions at San Jose State University, JPL/NASA, Xerox PARC and Pyramid/Siemens Technology. Performance Dynamics has also embarked on joint research into Quantum Information Technology. Dr. Gunther is a member of the AMS, APS, ACM, CMG, IEEE, and INFORMS.

References

[1] J. Peek, T. O'Reilly, and M. Loukides. *UNIX Power Tools*. O'Reilly, Sebastopol, CA, 2nd edition, 1997.

[2] K. Hess. "Monitoring Linux Performance with Orca". `http://www.linux-magazine.com/issue/65/Linux_Performance_Monitoring_With_Orca.pdf`, 2006.

[3] N. Gunther. *Analyzing Computer System Performance Using Perl::PDQ*. Springer-Verlag, 2005.

[4] N. Gunther. "Berechenbare Performance". `http://www.linux-magazin.de/technical_review/technical_review_02_monitoring`, 2007.