Load Average enträtselt und erweitert

Leistungsdiagnostik

Shellkommandos wie »uptime« werfen stets drei Zahlen als Load Average aus. Allerdings wissen nur wenige, wie sie zustande kommen und was genau sie bedeuten. Dieser Beitrag klärt darüber auf und stellt zugleich mit dem Stretchfaktor eine Erweiterung vor. Neil Gunther

Etliche Kommandos, darunter auch »uptime«, »procinfo« »top« und »ruptime«, geben eine Kombination von drei Werten aus, die wohl jeder schon mal gesehen hat, der mit einem Linux-System arbeitet. Im Fall von »uptime« sieht das beispielsweise so aus:

```
21:36:04 up 43 min,  3 users, ➥
load average: 0.24, 0.19, 0.19
```

Doch so bekannt diese Load-Average-Werte sind, so oft bleiben sie dennoch unverstanden. Die Verwirrung beginnt schon bei dem Wort Load, das ganz verschiedene Bedeutungen hat. Ein Systemadministrator versteht darunter gewöhnlich die Anzahl aktiver User eines Systems, wogegen ein Performance-Analyst dabei eher die Auslastung des Servers im Auge hat. Average lässt sich einfach mit Durchschnitt übersetzen.

Die Manpage von »uptime« erläutert: „Uptime produziert eine einzeilige Ausgabe mit dem Load Average der letzten 1, 5 und 15 Minuten.“ Das erklärt zumindest, warum es drei Werte sind. Die Manpage von »procinfo« führt weiter aus, der Load Average sei „die durchschnittliche Anzahl laufender und ausführbarer Prozesse“. Eine noch weiter gehende Erklärung gibt ein Experte: „Der Load Average versucht die Anzahl der aktiven Prozesse zu einem bestimmten Zeitpunkt zu messen. Als ein Maß für die Auslastung der CPU ist er jedoch schlecht definiert und vereinfacht, dabei aber trotzdem nicht nutzlos.“ [1]

Das klingt nicht sehr ermutigend. In der Tat stellt sich heraus, dass der Load Average deshalb ein schlechtes Maß für die CPU-Auslastung ist, weil er diese gar nicht misst. Jedenfalls scheint der Begriff Load aber dafür irgendwie mit der Anzahl aktiver Prozesse zusammenzuhängen. Doch was wäre dann eine durchschnittliche Last? Der Experte beantwortet die Frage mit einer Gegenfrage: „Was ist hoch? Das hängt gewöhnlich vom System ab. Ideal ist ein Load Average von sagen wir drei. Definitiv hoch wäre er, wenn man »uptime« nicht mehr bräuchte, um zu erkennen, dass das System stark überlastet ist.“

Fragt sich nur, wieso ausgerechnet alles unter drei ein guter Wert ist? Die Erklärung fährt fort: „Verschiedene Systeme verhalten sich unter demselben Load Average unterschiedlich. Ein einzelner CPU-gebundener Background-Prozess kann die Antwortzeiten drastisch verlängern, obwohl der Load Average dabei ziemlich niedrig bleibt.“ Diese letzte Aussage stimmt tatsächlich.

Moderne grafische Tools wie Orca [2] eröffnen eine umfassendere Perspektive, indem sie den Load Average als Zeitreihe darstellen (Abbildung 1). Blair Zajac, der Entwickler von Orca, erklärt: „Wenn der Langzeit-Trend ansteigt und sich die Last nicht verlagern lässt, dann sind mehr oder schnellere CPUs nötig. Für eine optimale CPU-Auslastung sollte der Maximalwert nicht höher sein als die Anzahl der CPUs im System.“

Warten ist nicht immer schlecht

Diese Aussage erkennt, dass womöglich mehrere Prozessoren die Run Queue bedienen. Unglücklicherweise legt sie aber gleichzeitig nahe, dass jede Form von Queueing (etwa Wartezeiten) schlecht ist. Wie dieser Beitrag weiter unten noch belegt, ist nichts weiter von der Wahrheit entfernt. Keine Wartezeiten mögen für eine Number-Crunching-Applikation wünschenswert sein, aber für kommerzielle Workloads ist ein moderates Maß an Queueing immer zu erwarten und auch wünschenswert. Das ist schließlich

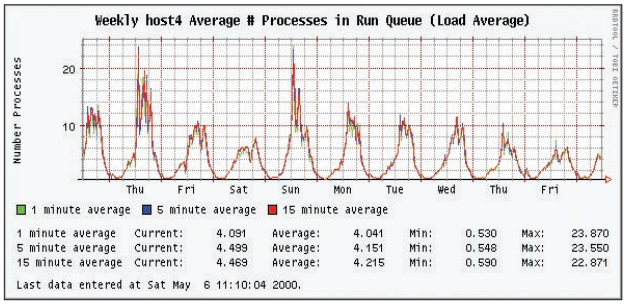

Abbildung 1: Periodische Proben des Load Average als einwöchige Zeitreihe. Die Werte für das 1-, 5- und 15-Minuten-Intervall sind hier überlagert und grün, blau und rot eingefärbt. Die grafische Darstellung übernahm Orca.

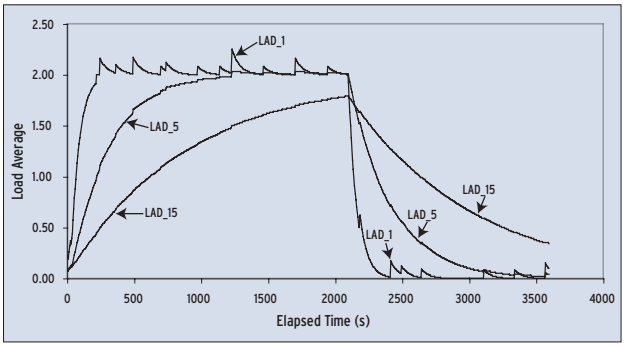

Abbildung 2: Die Load-Average-Daten (LAD) aus dem kontrollierten Experiment, LAD_1, LAD_5 und LAD_15 entsprechen dem 1-, 5- und 15-Minuten-Intervall.

der Grund dafür, dass Linux überhaupt eine Run Queue hat. Weitere Klärung sollen zwei Experimente auf einem Ein-Prozessor-System bringen.

Kontrollierte Experimente

Die Workload-Werte für dieses Experiment kamen im Verlauf einer Stunde (3600 Sekunden) auf einem ansonsten unbeschäftigten Ein-Prozessor-Linux-System zusammen. Der Test bestand aus zwei Phasen:

- Zwei CPU-intensive Jobs liefen für 2100 Sekunden im Hintergrund.
- Danach stoppten beide Prozesse gleichzeitig, die Aufzeichnung des Load Average lief aber noch 1500 Sekunden weiter. Das Perl-Skript aus Listing 1 steuerte das Experiment.

Die Last erzeugte ein C-Programm namens »burncpu.c«, das darauf ausgelegt ist, CPU-Zyklen zu verschwenden. Die Ausgabe von »top« während der Laufzeit von »getload.pl« sieht die beiden »burncpu«-Prozesse als die größten Rechenzeit-Verbraucher an erster Stelle der Liste (siehe Kasten „Ausgabe von »top« …“ auf der nächsten Seite).

Abbildung 2 zeigt, wie der Wert für das Minutenintervall nach etwa 300 Sekunden einen Load-Average-Wert von zwei erreicht. Der Wert für das 5-Minuten-Intervall kommt nach rund 1200 Sekunden dort an, während der 15-Minuten-Load-Average den Wert zwei nach etwa 4500 Sekunden erreicht. Die beiden Prozesse liefen dabei aber nicht länger als 2100 Sekunden.

Wer über ein wenig Hintergrundwissen in Elektrotechnik verfügt, dem fällt sicherlich sofort die deutliche Ähnlichkeit der Kurve in der Abbildung 2 mit dem Spannungsverlauf beim Laden und Entladen einer Widerstand-Kondensator-Schaltung (RC-Glied) auf. Wie sich zeigen wird, ist diese Ähnlichkeit auch nicht zufällig.

Interessant ist, dass der Maximalwert des Load Average hier gleich der Anzahl CPU-intensiver Prozesse ist. Die Zacken in der Kurve rühren von verschiedenen Daemons her, die zeitweilig aufwachten und sich wieder schlafen legten. Im Zusammenhang dieses Tests lassen sie sich als Hintergrundrauschen ansehen.

Nebenbei: Wäre nur ein Prozess gelaufen, wäre der Load Average nicht über einen Wert von eins gestiegen und der verzeihliche, aber dennoch falsche Schluss hätte nahe gelegen, dass er ein direktes Maß der CPU-Auslastung sei. Der Autor dieses Artikels erwähnt dies, weil er in einem anderen Zusammenhang selbst einmal diesen Fehler beging. ►

Listing 1: Mess-Skript »getload.pl«

```
01 #! /usr/bin/perl -w
02 $sample_interval = 5; # seconds
03
04 # Fire up background cpu-intensive tasks ...
05 system("./burncpu &");
06 system("./burncpu &");
07 # Perpetually monitor the load average via
   uptime and emit it as tab-separated
08 # fields for possibleuse in a
09 # spreadsheet program.
10
11 while (1) {
12     @uptime = split (/ /, 'uptime');
13     foreach $up (@uptime) {
14         # collect the timestamp
15         if ($up =~ m/(\d\d:\d\d:\d\d)/) {
16             print "$1\t";
17         }
18         # collect the three load metrics
19         if ($up =~ m/(\d{1,}\.\d\d)/) {
20             print "$1\t";
21         }
22     }
23     print "\n";
24     sleep ($sample_interval);
25 }
```

Listing 2: »calc_load()«

```
1136  unsigned long avenrun[3];
1137
1138  EXPORT_SYMBOL(avenrun);
1139
1140 /*
1141  * calc_load - given tick count, update the
      avenrun load estimates.
1142  * This is called while holding a write_
      lock on xtime_lock.
1143 */
1144  static inline void calc_load(unsigned long
       ticks)
1145 {
1146     unsigned long active_tasks;
      /* fixed-point */
1147     static int count = LOAD_FREQ;
1148
1149     count -= ticks;
1150
1151     if (unlikely(count < 0)) {
1152             active_tasks = count_active_
       tasks();
1153             do {
1154                     CALC_LOAD(avenrun[0],
      EXP_1, active_tasks);
1155                     CALC_LOAD(avenrun[1],
      EXP_5, active_tasks);
1156                     CALC_LOAD(avenrun[2],
      EXP_15, active_tasks);
1157                     count += LOAD_FREQ;
1158             } while (count < 0);
1159     }
1160 }
```

Ausgabe von >>top<< während des Experiments

```
  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
```

Als nächstes Ziel ist herauszufinden, warum die Load-Average-Kurve sich so verhält, wie Abbildung 2 zeigt. Dafür gilt es, Kernelcode zu studieren. Die Wahl fiel auf die Quellen der Release 2.6.20.1, die unter [3] verfügbar sind, komplett mit Cross-Referenzen und Hyperlinks für leichtere Lesbarkeit.

Zeitfrage

In den Quellen findet sich im Code für den CPU-Scheduler die Funktion »calc_load()« (Listing 2) im File unter [http://lxr.linux.no/source/kernel/timer.c]. Es handelt sich um die Routine, die direkt den Load Average berechnet. Zuerst prüft sie, ob die Sample-Periode abgelaufen ist, setzt den Sampling-Zähler zurück und ruft danach das Unterprogramm »CALC_LOAD()« auf, um den 1-, 5- und 15-Minuten-Wert zu berechnen. Die verwendete Sampling-Periode »LOAD_FREQ« beträgt »5*HZ«.

Der Hintergrund: Jeder Linux-Rechner enthält eine Hardware-Uhr. Diese Uhr tickt mit einer festen Frequenz, mit der sich alle anderen Komponenten des Systems synchronisieren. Damit die Komponenten sich auf die Schlagfrequenz der Uhr einstellen können, sendet sie einen Interrupt mit jedem Tick. Das tatsächliche Intervall zwischen den Ticks ist plattformspezifisch, bei den meisten Linux-Systemen beträgt es 10 Millisekunden. Diese spezifische Taktrate speichert das System in einer Konstanten namens »HZ«, die das Header-File »param.h« enthält. In den hier zugrunde liegenden Onlinequellen findet sich der Wert 100:

```
# define        HZ      100
```

Jede Zeit-Sekunde teilt sich also in 100 Ticks. Mit anderen Worten: Mit jedem Tick, also jede Hundertstelsekunde oder alle 10 Millisekunden, löst die Uhr einen Interrupt aus. Das Makro in Zeile 73 konvertiert die Anzahl der Ticks wieder zurück in Sekunden:

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

Die mit »HZ« benannte Konstante ist also der Frequenz-Teiler und nicht etwa die SI-Einheit Hertz. Letztere hätte übrigens auch die Abkürzung Hz. Also meint »5 * HZ«: 500 Ticks oder 500 mal 10 Millisekunden oder 5 Sekunden.

Daraus ergibt sich, das »CALC_LOAD()« alle 5 Sekunden läuft – und nicht etwa fünfmal pro Sekunde, wie man irrtümlich annehmen könnte. Auch ist darauf zu achten, diese Sampling-Periode nicht mit den Berichtsperioden von 1, 5 und 15 Minuten zu verwechseln.

Das C-Makro »CALC_LOAD()« leistet die eigentliche Arbeit beim Berechnen des Load Average. Es tut dies mit dem in Listing 3 zu sehenden Codefragment, das sich unter [http://lxr.linux.no/source/include/linux/sched.h] findet. Der Code wirft aber unweigerlich eine Reihe von Fragen auf:

- Wo kommen die merkwürdigen Zahlen 1884, 2014 und 2037 her?
- Welche Rolle spielen sie bei der Berechnung des Load Average?
- Was macht »CALC_LOAD()« tatsächlich?

Damit sich diese Fragen beantworten lassen, ist ein kleiner Ausflug in die Festkomma-Welt nötig.

Festkomma-Arithmetik

Die kryptischen Kommentare in »CALC_LOAD()« erklären bereits, dass das Makro Festkomma- statt Gleitkommazahlen zur Berechnung des Load Average verwendet. Weil die Berechnungen im Kernel stattfinden, gingen die Entwickler wohl davon aus, das Festkomma-Rechnungen effizienter sind.

Eine Festkomma-Darstellung verwendet eine begrenzte Anzahl von Stellen zur Darstellung jeder Zahl, auch für Zahlen mit einem gebrochenen Teil hinter dem Dezimalkomma (Mantisse). Mit vier Stellen Genauigkeit lassen sich so Zahlen wie 0,123, -12,3401 oder 1234,0001 exakt wiedergeben, wogegen Zahlen wie 0,12346 oder -8,34051 nicht exakt darstellbar sind und generell gerundet als 0,1235 und -8,3405 erscheinen.

Wie der Kommentar (ab Zeile 103) warnt, können zu viele aufeinander folgende Rundungen dazu führen, dass sich sonst nicht signifikante Fehler in signifikante verwandeln. Ein Ausweg besteht darin,

Listing 3: Makro »CALC_LOAD()«

```
 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;
```

Tabelle 1: Bitpositionen für das 10,11-Format

<- 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

die Anzahl der Stellen zu erhöhen, die die Mantisse darstellen, und zwar in der Annahme, dass mehr Speicherplatz auch eine größere Genauigkeit garantiert.

Das M,N-Format

Der Kommentar weist darauf hin, dass die Funktion 10 Bits für den Ganzzahl-Teil und 11 Bits für den gebrochenen Anteil verwendet. Das nennt sich auch M,N = 10,11-Format. Die Regeln für die Festkomma-Addition sind dieselben wie für Gleitzahl-Addition. Der große Unterschied ergibt sich bei der Multiplikation. Das Produkt zweier Festkommazahlen ergibt sich als:

$$M,N \bullet M,N = (M+M),(N+N) \qquad (1)$$

Um das Ergebnis wieder ins M,N-Format zu konvertieren, eliminiert die Rechnung die niederwertigen Bits durch Verschieben um N Bits.

Die »CALC_LOAD()«-Routine verwendet diverse Festkomma-Konstanten. Die erste ist die 1 selber, hier »FIXED_1« bezeichnet. In Tabelle 1 entspricht die erste Zeile »FIXED_1«, wogegen in der Zeile darunter die führenden Nullen und das Dezimalkomma entfernt wurden. Die Ziffern der so entstandenen Binärzahl 100000000000_2 indiziert die unterste Tabellenzeile. Daraus ergibt sich, dass »FIXED_1« das Äquivalent von 2^{11} ist oder 2048 in der vertrauten Dezimalschreibweise.

Benutzt man den C-Operator für bitweises Linksverschieben (»<<«), lässt sich 100000000000_2 als »1 << 11« schreiben, in Übereinstimmung mit Zeile 111 des Makros. Alternativ ließe sich »FIXED_1« als dezimale Ganzzahl

$$FIXED\_1 = 2048_{10} \qquad (2)$$

darstellen, um die Berechnung der verbleibenden »EXP_1«-, »EXP_5«- und »EXP_15«-Konstanten für das 1-, 5- und 15-Minuten-Intervall zu vereinfachen. Als Beispiel diene das 1-Minuten-Intervall. Die Sample-Periode sei σ und die Reporting-Periode *r*. Dann ergibt sich:

$$EXP\_1 \equiv e^{-\sigma/r} \qquad (3)$$

Die Werte von σ = 5 Sekunden sowie *r* = 60 Sekunden sind bereits bekannt. Damit errechnet sich der Dezimalwert von EXP_1 aus:

$$e^{-5/60} = 0,920044414643 \qquad (4)$$

Um (4) in das 10,11-Format zu konvertieren, ist das Ergebnis nur mit der Konstanten »FIXED_1« oder mit 1 zu multiplizieren und anschließend auf die nächste 11-Bit-Ganzzahl zu runden. Auf diese Weise erhält man:

$$[2048 \bullet 0,92004441463] = 1884_{10} \qquad (5)$$

Jede der magischen Zahlen lässt sich in dieser Weise berechnen. Das Ergebnis fasst Tabelle 2 zusammen. Die Resultate dort stimmen mit dem Kernel überein, der festlegt:

Tabelle 2: Magische Zahlen für 5-Sekunden-Sampling

Basis	Sekunden	$e^{-5/r}$	gerundet	binär
r_1	60	1884,25	1884_{10}	11101011100_2
r_5	300	2014,15	2014_{10}	11111011110_2
r_{15}	900	2036,65	2037_{10}	11111110101_2

Tabelle 3: Magische Zahlen für 2-Sekunden-Sampling

Basis	Sekunden	$e^{-2/r}$	gerundet	binär
r_1	60	1980,86	1981_{10}	11110111101_2
r_5	300	2034,39	2034_{10}	11111110010_2
r_{15}	900	2043,45	2043_{10}	11111111011_2

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

Würde man die Sampling-Periode auf 2 Sekunden verkürzen, so ändern sich die Konstanten so, wie es Tabelle 3 zeigt. Das erklärt die Herkunft der drei magischen Zahlen. Als Nächstes ist nachzuvollziehen, wie das »CALC_LOAD()«-Makro tatsächlich rechnet.

Load Average enträtselt

Mathematisch betrachtet nimmt »CALC_LOAD()« den aktuellen Wert der Variablen »load« und multipliziert ihn mit dem Faktor »exp«. Zu diesem Wert addiert es dann einen Term, der die Anzahl der aktuellen Prozesse »n« enthält, multipliziert mit einem weiteren Term »FIXED_1-exp«. Die letzte Zeile des Makros konvertiert das Ergebnis ins Dezimalformat.

Aufgrund (3) ist bereits bekannt: Die Makro-Variable »exp« entspricht $e^{-\sigma/r}$ und wegen (2) und (3) ist »FIXED_1« äquivalent zu $1\text{-}e^{-\sigma/r}$. In einer etwas gebräuchlicheren mathematischen Form aufgeschrieben sieht das Makro »CALC_LOAD()« so aus:

$$L(t) = L(t-1)\, e^{-\sigma/r} + n(t)\,(1 - e^{-\sigma/r}) \qquad (6)$$

Darin sind »L(t)« der aktuelle Wert der Load-Variablen, »L(t-1)« der Wert aus der vorherigen Sample-Periode und »n(t)« die Anzahl aktiver Prozesse.

Ein Linux-Prozess kann sich in einem von ungefähr einem halben Dutzend Zuständen befinden (je nachdem, wie man zählt), von denen Running, Runable (»R« in der Ausgabe von »ps«) und Sleeping (»S«) die wichtigsten sind. Eine hübsche Animation der Status und der Übergänge zwischen ihnen findet sich bei [4]. Die Load-Average-Metrik basiert auf der Gesamtzahl

- der ausführbaren Prozesse in der Run Queue des Schedulers und
- der gegenwärtig durch den Prozessor ausgeführten Prozesse. ▶

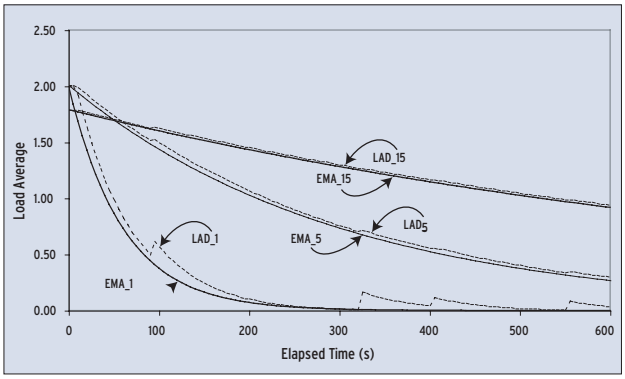

Abbildung 3: Vergleich der beobachteten Load-Average-Daten (LAD) mit den nach (8) berechneten Daten (EMA) für 600 Sekunden nach Terminierung der Prozesse. Vorausberechnete und gemessene Werte stimmen gut überein.

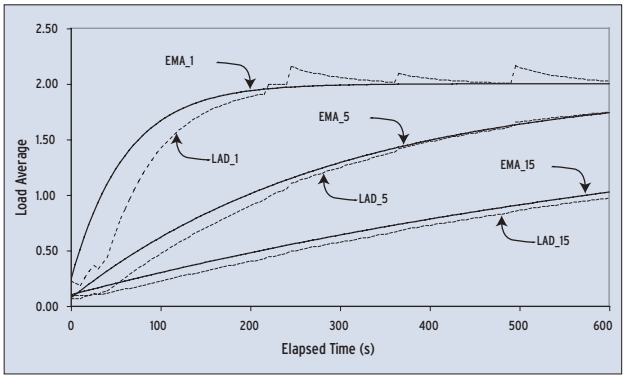

Abbildung 4: Vergleich der beobachteten Load-Average-Daten (LAD) mit den nach (9) berechneten (EMA) während der Messperiode 0 <= T <= 600 Sekunden aus Abbildung 2. Auch hier ist eine gute Übereinstimmung zu erkennen.

Das ist die korrekte Definition der kryptischen Umschreibung „aktiver Prozess" in der eingangs zitierten Erklärung des Experten. Die Queueing-Theorie bezeichnet diese aktiven Prozesse zusammen als Queue und meint damit nicht nur die in der Warteschlange (der Run Queue des Prozessors), sondern auch die gegenwärtig laufenden. »CALC_LOAD()« ist also nichts anderes als die Festkommaversion von (6). Die Gleichung (6) wiederum erklärt sich am besten anhand einiger Spezialfälle.

Leere und volle Run Queue

Zuerst sei der Fall betrachtet, in dem die Run Queue leer ist, es also weder einen laufenden noch ausführbare Prozesse gibt. Mit $n(t) = 0$ in Gleichung (6) ergibt sich:

$$L(t) = L(t-1)\, e^{-\sigma/r} \qquad (7)$$

Beim Iterieren von (7) zwischen $t = t_0$ und $t = T$ erhält man:

$$L(T) = L(t_0)\, e^{-\sigma t/r} \qquad (8)$$

Den Grafen der Gleichung (8) zeigt Abbildung 3 für die drei Load-Average-Metriken (r). Er korrespondiert mit einem zeitabhängigen exponentiellen Abklingen. Mit anderen Worten: Diese Berechnung entspricht dem beobachteten Abfall des Load Average zwischen $t_0 = 2100$ und $T = 3600$ aus Abbildung 2. Abbildung 3 vergleicht die gemessenen Daten (Label LAD) mit den nach Gleichung (8) berechneten (Label EMA).

Der zweite Spezialfall entspricht dem Anfangsstadium des Experiments mit einer Run Queue, die ständig von zwei Prozessen belegt ist. Nun dominiert der zweite Term aus Gleichung (6) und eine Iteration von $t = t_0$ und $t = T$ ergibt:

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

In gleicher Weise wie eben zeigt nun Abbildung 4 wieder die gemessenen (LAD) und die berechneten Werte (EMA) für die drei Load-Average-Metriken (r). Es ergibt sich eine monoton ansteigende Funktion und erkennbar ist, dass (9) verantwortlich für den Anstieg zwischen $t_0 = 0$ und $T = 2100$ in Abbildung 2 ist.

Elektrotechnik-Analogie

Nachdem bereits auf die Ähnlichkeit zum Spannungsverlauf in einem RC-Glied hingewiesen wurde, lässt sich diese Analogie nun einen Schritt weiterführen. Aus der Schaltungstheorie ist bekannt, dass die Anstiegszeit ungefähr das Fünffache der charakteristischen Zeitkonstanten r beträgt. In »CALC_LOAD()« ist $r_1 = 60$ Sekunden. Daraus würde sich eine Anstiegszeit von ungefähr $5r_1$, also rund 300 Sekunden ergeben – und das ist genau der Fall, wie Abbildung 4 zeigt. Die anderen Anstiegszeiten fasst Tabelle 4 zusammen.

Dieses Verhalten legt nahe, dass die Art und Weise, nach der sich der Load Average berechnet, gar nichts Neues ist. Tatsächlich handelt es sich um eine gewöhnliche Technik für die Aufbereitung hochvariabler Rohdaten für die darauf folgende Analyse: Die Daten durchlaufen eine Glättungsfunktion. Die generelle Beziehung zwischen den Rohdaten der Eingabe und der geglätteten Ausgabe ist gegeben durch:

$$\underbrace{Y(t)}_{\text{geglättet}} = Y(t-1) + \underbrace{\alpha}_{\text{const.}}\big[\underbrace{X(t)}_{\text{Rohdaten}} - Y(t-1)\big] \qquad (10)$$

Die Glättungsfunktion in (10) ist ein Exponentialfilter oder ein exponentiell geglätteter gleitender Durchschnitt (Exponentially-smoothed Moving Average, EMA) von der Art, wie er beispielsweise auch bei Finanz-Vorhersagen [5] oder in der Signalverarbeitung Verwendung findet. Der Parameter α in Gleichung (10) nennt sich gewöhnlich die Glättungskonstante, wogegen $1-\alpha$ als Dämpfungsfaktor bezeichnet wird. Überdies lassen sich beide Faktoren direkt zu den korrespondierenden Faktoren aus (6) in Beziehung bringen [6]. Die Größe des Glättungskoeffizienten $0 <= \alpha <= 1$ bestimmt, wie stark die gegenwärtige Voraussage aufgrund des Fehlers in der vorherigen Iteration zu korrigieren ist.

Zu beachten ist, dass der exponentielle Dämpfungsfaktor für r_q in Tabelle 6 mit dem Wert aus Gleichung (4) bis auf vier Dezimalstellen übereinstimmt. Der 1-Minuten-Load-Average hat die geringste Dämpfung oder rund acht Prozent Kor-

Tabelle 4: Anstiegszeiten für die Daten aus Abbildung 4

Load-Average-Parameter	Zeitkonstante	Erwartete Anstiegszeit
r_1	60	300 Sekunden
r_5	300	1500 Sekunden
r_{15}	900	4500 Sekunden

Tabelle 5: Stretchfaktor-Definition	
m	Anzahl der Prozessoren oder Cores
Q	gemessener Load Average
p	gemessene Prozessor-Auslastung

rektur. Er reagiert am empfindlichsten auf unmittelbare Änderungen in der Länge der Run Queue. Dagegen hat der 15-Minuten-Load-Average die höchste Dämpfung oder nur ein Prozent Korrektur und reagiert am trägsten.

Stretchfaktor

Was ist ein guter Wert für den Load Average? Ungeachtet der anfangs zitierten Expertenmeinung, derzufolge drei ein guter Wert für den Load Average sei, entsteht der Wunsch, eine Daumenregel zu konstruieren, die den Load Average auch auf Multicore- und Multiprozessorsystemen quantitativ beurteilt. Mehr noch, nachdem bekannt ist, dass der Load Average ein exponentiell gedämpfter, gleitender Durchschnitt der Aktivitäten in der Run Queue des Prozessors ist, lässt sich diese Frage umwandeln in: Wie lang sollte diese Queue sein?

Wie alle Performance-Fragen, sieht auch diese nur einfach aus. Tatsächlich ist die Frage nach dem idealen Load-Average-Wert falsch gestellt (ill-posed). Dummerweise ähnelt sie der Betrachtung: Wie lang ist ein Stück Seil? Mit der leicht zynischen Antwort: Nicht lang genug, um sich als unerwünschte Folge damit strangulieren zu können. Auch Queues sollten nicht so lang sein, dass sich unerwünschte Folgen einstellen.

Lange Queues korrespondieren mit langen Antwortzeiten, weshalb die Aufmerksamkeit tatsächlich der Metrik Antwortzeit gehören muss. Denn ab wann eine lange Queue schlechte Antwortzeiten hervorruft, hängt davon ab, was der Anwender als schlecht empfindet.

Den meisten Performance-Management-Tools fehlt diese Verbindung von der Länge der Run Queue zu den von den Nutzern registrierten Antwortzeiten. Ein anderes Problem besteht darin, dass die Länge der Queue ein absolutes Maß ist, man aber eigentlich ein relatives Maß braucht. Denn auch die subjektiven Einschätzungen wie „gut“ und „schlecht“ sind relativ.

Ein solches relatives Maß ist der Stretchfaktor der die mittlere Länge der Run Queue relativ zur durchschnittlichen Anzahl der in Bearbeitung befindlichen Jobs misst. Der Stretchfaktor gibt Vielfache von Bedieneinheiten (Service Units) an. Ein Wert von eins bedeutet, dass keine Wartezeit anfällt.

Was den Stretchfaktor wirklich nützlich macht, ist der Umstand, dass er leicht mit den Zielvorgaben von Service Level Agreements (SLAs) vergleichbar ist. Sie definieren Ziele (Service Level Objectives, SLOs) oft als Vielfache bestimmter Geschäftseinheiten, etwa Angebote pro Stunde bei einer Versicherung. Eine solche Zielvorgabe könnte dann konkret etwa lauten: Die durchschnittliche Antwortzeit für den Benutzer soll während der höchsten Belastung zwischen 10 und 12 Uhr die Anzahl von 15 Service-Einheiten nicht übersteigen. Das entspricht der Aussage, das SLO solle den Stretchfaktor 15 nicht überschreiten.

Mit den in Tabelle 5 definierten Symbolen berechnet sich der Stretchfaktor f auf folgende Weise:

$$f = \frac{Q}{mp} \qquad (11)$$

Aus dem oben beschriebenen kontrollierten Experiment sind die Werte von m = 1 (Ein-Prozessor-System), Q = 2 für den 1-Minuten-Load-Average und p = 1 bekannt, denn der Workload war CPU-gebunden. Wer diese Werte in die Gleichung (11) einsetzt, erhält das Resultat f = 2. Es besagt, dass der Erwartungswert für die Abarbeitung jedes Prozesses bei zwei Serviceperioden liegt.

Wie lang eine solche Serviceperiode – in Zeiteinheiten gemessen – tatsächlich ist, spielt dafür keine Rolle. In diesem Fall haben Load Average und Stretchfaktor einmal den gleichen Wert, weil die Prozesse auf einem Prozessor laufen und CPU-intensiv sind.

Tabelle 6: Dämpfungsfaktoren für »CALC_LOAD()«		
Zeitbasis	Dämpfungsfaktor	Glättungskonstante
r_1	0,9200	0,0800 (rund 8%)
r_5	0,9835	0,0165 (rund 2%)
r_{15}	0,9945	0,0055 (rund 1%)

Tabelle 7: Spam-Server, Tagesstatistik	
Anzahl CPUs	4
Spam erkannt	33 901
Ham akzeptiert	23 123
E-Mails verarbeitet	57 024
E-Mails pro Stunde	2376
Per CPU und Stunde	594
CPU busy	99 Prozent
Sekunden pro E-Mail	6
Load Average	97,36

Antispam-Farm und Number Cruncher

Im Folgenden sollen einige Beispiele aus der Praxis zeigen, wie der Stretchfaktor einsetzbar ist. Ein gut bekanntes und besuchtes Webportal mag aus einigen hundert Servern bestehen. Alle großen Webhoster verwenden Spamfilter für ihre E-Mail-Kunden. Eine typische Konfiguration könnte aus einem Satz Server bestehen, auf denen ein Filter-Tool wie Spamassasin läuft. Es seien zwei Dualcore-Maschinen vorhanden, die rund um die Uhr Mails scannen. Eine typische Tagesstatistik sieht beispielsweise so aus wie in Tabelle 7.

Ein Load Balancer soll die Arbeit in der Serverfarm möglichst effektiv verteilen. Die Arbeit des Load Balancers überwachen die Admins mit Hilfe des 1-Minuten-Load-Average. Eine Probe mit dem Load Average von 50 Servern demonstriert Abbildung 5. Deutlich zeigt sich ein Ungleichgewicht. Daraus ergeben sich für den Administrator vor allem fünf wichtige Fragen:

1. Woher kommt das Ungleichgewicht?
2. Sind die meisten Server als Folge der ungleichen Verteilung überlastet?
3. Ist ein Load Average von 97,36 akzeptabel?
4. Welche Performance würde tatsächlich gebraucht?
5. Wie viele zusätzliche Server wären erforderlich, um die gegenwärtige Scan-Performance im nächsten Jahr bei gestiegenem Mail-Aufkommen auf dem gleichen Niveau zu halten? ▶

Abbildung 5: Proben des Load Average von 50 Servern einer Beispiel-Farm. Das Diagramm lässt eine Ungleichverteilung durch den Load Balancer deutlich erkennen.

Der Stretchfaktor im Beispiel ergibt sich nach dem Einsetzen der Werte in die Gleichung (11). Die Rechnung sieht dabei so aus:

$$f = \frac{97,36}{4 \bullet 0,99} = 24,59 \qquad (12)$$

Aus Tabelle 7 ist ein durchschnittlicher Zeitbedarf für das Scannen einer E-Mail von ungefähr sechs Sekunden zu entnehmen. Demnach besagt ein Stretchfaktor von rund 25 jetzt konkret, dass es $25 \cdot 6 = 150$ Sekunden oder reichlich zwei Minuten dauert, bis eine Mail, die das Portal von außen erreicht, komplett auf Spam überprüft und anschließend in die Mailbox ihres Adressaten eingeordnet worden ist.

Ein Absolutwert wie Q = 97,36 sagt dagegen herzlich wenig. Der relative Stretchfaktor andererseits gibt an, wie viele Serviceperioden das Spamfiltern tatsächlich kostet. Die Antwort auf Frage 3, inwieweit nämlich der zuständige Admin mit dem Load-Average-Wert zufrieden sein kann, hängt jetzt einzig und allein von den vereinbarten Servicezielen ab. Durch einen solchen Vergleich der Soll- und Ist-Werte lässt sich die Frage auch tatsächlich quantitativ und nicht nur spekulativ beantworten.

Die Daten aus Tabelle 7 können aber genauso gut dazu dienen, mit Hilfe eines Vorhersage-Tools wie PDQ die Frage 4 zu beantworten. Wesentlich weiter führende Informationen zu PDQ enthalten der Kasten „PDQ in Python“ sowie [6] und [7].

Das oben schon beschriebene Spam-Server-Modell sähe abgebildet in einem PDQ-Modell so aus, wie es Listing 4 demonstriert. Führt man PDQ mit diesem Modell aus, dann produziert die Berechnung die folgenden Ergebnisse:

```
****** 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
```

Der von PDQ berechnet Stretchfaktor ist hier etwas höher als die Berechnung nach Gleichung (12) ergeben hatte. Die Ursache klärt ein Blick auf den Abschnitt des PDQ-Reports, der die Performance-Information enthält:

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

Bei der hohen Frequenz, mit der ständig neue Arbeit eintrifft (2376 Mails pro Stunde), sollte jede CPU zu 99 Prozent ausgelastet sein. Dieser Wert ist in der Praxis aber wegen des Ungleichgewichts beim Load Balancing nicht erreichbar. PDQ geht jedoch von einer idealen Lastverteilung aus, daher liegt der vorhergesagte Load Average näher an 100 E-Mails und der Stretchfaktor ist mit 25,45 in der Praxis etwas höher als der vorausberechnete Wert von 24,50.

Frage 5 zielt auf Zukunftsplanungen. Selbst wenn der Admin den gegenwärtigen Stretchfaktor unter Spitzenlast akzeptiert, laufen doch alle Server nahe an der Sättigungsgrenze. Um höhere Belastungen abzufangen, wären also schnellere CPUs oder – wahrscheinlicher – zusätzliche Vier-Wege-Server erforderlich. PDQ kann dabei helfen, die nötige Anzahl neuer Server für einen vorgegebenen Stretchfaktor zu berechnen.

Geodaten

Das nächste Beispiel nutzt ein ähnliches PDQ-Modell, um zu zeigen, was passiert, wenn es gar keine Warteschlange trotz beschäftigter CPUs gibt. Die Annahme ist, dass jeder Linux-Prozess 10 Stunden für seine Abarbeitung braucht,

PDQ in Python

PDQ (Pretty Damn Quick) ist ein Modellierungstool für die Analyse der Performance von Rechner-Ressourcen wie Prozessoren, Platten oder Gruppen von Prozessen, die diese Ressourcen beanspruchen. Ein PDQ-Modell wird mit Hilfe von Algorithmen aus der Queueing-Theorie analysiert. Die aktuelle Release erlaubt das Erstellen solcher Performance-Modelle in C, Perl, Python, Java und PHP.

Die Beispiele dieses Artikels benutzen Funktionen wie folgende:

- »pdq.Init()« initialisiert interne PDQ-Variable.
- »pdq.CreateOpen()« erzeugt einen Workload.
- »pdq.CreateNode()« erzeugt einen Server.
- »pdq.SetDemand()« setzt die Workload-Servicezeit der Server-Ressource.
- »pdq.Solve()« berechnet Performance-Metriken
- »pdq.Report()« erzeugt einen generischen Report.

Weitere Informationen finden sich auf der Website [www.perfdynamics.com/Tools]
Überblick: [... com/PDQ.html]
Download: [... com/PDQcode.html]
Manual: [... com/PDQman.html]
Den Autor dieses Artikels und Peter Harding entwickeln und pflegen PDQ.

weil er beispielsweise komplexe Daten aus der Erdölerkundung für Geophysiker aufbereitet. Das PDQ-Modell für diesen Fall zeigt Listing 5. Der zugehörige PDQ-Report sieht so aus:

```
****** RESOURCE Performance *******
Metric Resource       Value      Unit
------ -------- ---- -----      ----
Throughput   ...      0.0990     Jobs/Hour
Utilization  ...     24.7500     Percent
Queue length ...      0.9965     Jobs
Waiting line ...      0.0065     Jobs
Waiting time ...      0.0656
```

Danach gäbe es keine Warteschlange und alle vier CPUs wären beschäftigt, aber nur zu 25 Prozent ausgelastet. In der CPU-Statistik des laufenden Systems ließen sich tatsächlich 100 Prozent Auslastung beobachten. Das zu verstehen hilft wieder der 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
```

Stretchfaktor ist eins, denn es gibt keine Warteschlange. Andererseits dauert ein Job 10 Stunden, sodass auch die Antwortzeit 10 Stunden beträgt.

Der Grund für die unerwarteten Ergebnisse liegt darin, dass PDQ von dem Verhalten des Systems im stabilen Zustand ausgeht (Steady State). Um zu erkennen, wie sich das System unter diesen Bedingungen verhält, wäre eine viel längere Beobachtungszeit nötig, 100 Stunden oder länger. Niemand muss das wirklich tun, aber PDQ sagt, wie die Dinge lägen, wenn man es täte.

Weil die durchschnittliche Serviceperiode relativ lange dauert, ist umgekehrt die Anforderungsrate (Request Rate) entsprechend niedrig, sodass sich keine Warteschlange bildet. Das wiederum heißt, dass die Prozessorauslastung mit 25 Prozent ebenfalls niedrig ist – auf lange Sicht. Wer das System dagegen nur wenige Minuten beobachtet, während es gerade die Erkundungsdaten berechnet, der sieht einen Schnappschuss, nicht den stabilen Zustand.

Die kontrollierten Experimente am Anfang und die beiden Beispiele Spamfilter und Geodaten-Aufbereitung zeigen CPU-gebundene Workloads, weil sich damit die Beziehungen der Metriken untereinander am besten illustrieren lassen. I/O-gebundene Workloads würden geringere Load Averages ergeben, weil das Betriebssystem solche Prozesse, während sie auf Daten warten, suspendiert oder schlafen legt. In diesem Zustand sind sie weder ausführbar noch werden sie ausgeführt, tragen also nichts zu n(t) in Gleichung (6) bei.

Wer jetzt zur Ausgangsfrage nach einer Daumenregel zurückkehrt, erkennt, dass Q als die Gesamtzahl der wartenden und ausgeführten Prozesse nicht sehr aussagekräftig ist. Kombiniert mit der Anzahl der Prozessoren m und ihrer durchschnittlichen Auslastung p dagegen, ergibt sich der Stretchfaktor f als bessere Metrik für symmetrische Multiprozessor- und Multicore-Server.

Fazit

Die Absicht des Load Average ist, Trends im Wachstum der Run Queue aufzuzeigen. Daher gibt er drei Werte aus, die drei Zeitpunkte in der Vergangenheit repräsentieren. Verglichen mit heutigen Möglichkeiten für die grafische Darstellung von Trends wirkt das Verfahren antiquiert. Tatsächlich gehört der Load Average zu den ältesten Formen der Betriebssystem-Instrumentierung. Seine Ahnengalerie beginnt bereits ungefähr 1965 mit CTSS und setzt sich über Multics und Unix bis zu Linux fort [8].

Der Load Average verwendet einen exponentiell gewichteten, gleitenden Durchschnitt (EMA) von Proben der Run Queue. Die Glättung vermeidet Ausreißereffekte und die Notwendigkeit, Probendaten im Kernel zu puffern. Schließlich stellt sich der Stretchfaktor als bessere Performance-Metrik heraus, die sich besonders für den direkten Vergleich mit Servicezielen anbietet. *(jcb)* ■

Infos

[1] J. Peek, T. O'Reilly and M. Loukides, „UNIX Power Tools": O'Reilly, Sebastopol, CA, 2nd Edition, 1997

[2] K. Hess, „Watching the Waters – Monitoring Linux Performance with Orca": [http://www.linux-magazine.com/issue/65/Linux_Performance_Monitoring_With_Orca.pdf]

[3] Kommentierte Kernelquellen: [http://lxr.linux.no/source/]

[4] Prozess-Zustände: [http://www.linux-tutorial.info/modules.php?name=Tutorial&pageid=84]

[5] Finanzvorhersagen: [http://bigcharts.marketwatch.com]

[6] N. Gunther, „Analyzing Computer System Performance Using Perl::PDQ": Springer-Verlag, 2005

[7] N. Gunther, „Berechenbare Performance": Linux Technical Review 02/07, S. 112

[8] Load Average Historie: [http://www.multicians.org/InstrumentationPaper.html]

Der Autor

Dr. Neil Gunther ist ein international anerkannter Performance-Experte und gründete 1994 die Firma Performance Dynamics Company. Zuvor forschte er an der San Jose State University und war bei JPL, der NASA, Xerox PARC und Pyramid/Siemens beschäftigt. Er ist Mitglied der AMS, APS, ACM, CMG, IEEE und von Informs.

Listing 4: PDQ-Spamfarm-Modell

```
01 #!/usr/bin/env python
02 import pdq
03 # Measured performance parameters
04 cpusPerServer = 4
05 emailThruput = 2376 # emails per hour
06 scannerTime = 6.0 # seconds per email
07 pdq.Init("Spam Farm Model")
08 # Timebase is SECONDS ...
09 nstreams = pdq.CreateOpen("Email",
   float(emailThruput)/3600)
10 nnodes = pdq.CreateNode("spamCan",
   int(cpusPerServer), pdq.MSQ)
11 pdq.SetDemand("spamCan", "Email", scannerTime)
12 pdq.Solve(pdq.CANON)
13 pdq.Report
```

Listing 5: PDQ-Modell Ölerkundung

```
01 #!/usr/bin/env python
02 import pdq
03 processors = 4 # Same as spam farm example
04 7
05 arrivalRate = 0.099 # Jobs per hour (very low
   arrivals)
06 crunchTime = 10.0 # Hours (very long service time)
07 pdq.Init("ORCA LA Model")
08 s = pdq.CreateOpen("Crunch", arrivalRate)
09 n = pdq.CreateNode("HPCnode", int(processors),
    pdq.MSQ)
10 pdq.SetDemand("HPCnode", "Crunch", crunchTime)
11 pdq.SetWUnit("Jobs")
12 pdq.SetTUnit("Hour")
13 pdq.Solve(pdq.CANON)
14 pdq.Report()
```