Performanceanalyse -
oder: Was macht eigentlich mein Solaris?

Jörg „c0t0d0s0.org“ Möllenkamp
Oracle B.V. & Co. KG
Hamburg

Schlüsselworte:

Solaris, Performance, Tuning, Administration

Einleitung
Schon herkömmliche Unix-Systeme haben eine erhebliche Anzahl von Werkzeugen, an denen man feststellen kann, was gerade eigentlich auf einem System passiert. Viele betrachten die Verfügbarkeit solcher Tools als einen wesentlichen Vorteil unixoider Betriebssysteme.

Solaris trieb das in der zehnten Iteration noch ein erhebliches Stück weiter. Mit dem in Solaris 10 integrierten DTrace werden wir gleichsam von den auf uns herabstürzenden Informationen erschlagen. Informationen zu bekommen ist somit kein Problem mehr, doch was ist davon eigentlich sinnvoll, und wie gehe ich eigentlich an das Problem "Tuning", wie gehe ich an das Problem „Bottlenecksuche“ heran.

Als ich diesen Vortrag eingereicht habe, dachte ich eigentlich an eine Einführung in die Tools, die bei Solaris dabei sind. Im Rahmen einiger Kundensituationen fiel mir allerdings auf, das eine Erklärung von Tuning auf Unix-Systemen grundlegender sein muss.

Zudem möchte ich anmerken, das vieles von dem, was im folgenden aufgezählt wird, so auch bei anderen Betriebssystemumgebungen richtig ist, auch wenn sich das Werkzeug teilweise erheblich unterscheiden kann, insbesondere wenn man den Bereich der unixoiden Betriebssysteme verlässt.

Allgemeine Gedanken zum Tuning

Kenne Unix!
Es ist unbedingt nötig zum Tuning eines System zu mindestens grundlegend zu wissen, was eigentlich ein Unix macht. Will man mehr als nur „common wisdom“ überprüfen, so ist es sehr hilfreich sich der Zusammenhänge in einem Unix-System bewusst zu werden, um beispielsweise die grundlegenden Begrifflichkeiten des „virtual memory systems“ zu kennen, um so einen minor von einem major page fault begrifflich unterscheiden zu können, um überhaupt sinnvoll die Daten verstehen zu können.

Ich möchte hier eine Literaturempfehlung aussprechen, da eine Erläuterung der gesamten Zusammenhänge sicherlich den Rahmen dieses Dokuments sprengen würde: „Solaris Internals: Solaris 10 and Open Solaris Kernel Architecture“[1] von Jim Mauro und Richard McDougall. Dieses Buch ist sehr hilfreich, um die umfangreichen Zusammenhänge innerhalb von Solaris zu verstehen, um so die Ausgaben des Systems in den statistischen Tools richtig einordnen zu können.

1 Jim Mauro, Richard McDougall - „Solaris Internals: Solaris 10 and Open Solaris Kernel Architecture“ - http://amzn.to/aAnAyK - ISBN 0131482092

Eigentlich ist alles normal!
Nehmen wir mal an, sie haben obiges Buch gelesen und wissen nun sehr viel über Solaris. Reicht das zum Tuning eines Systems? Reicht dies zur Abschätzung, wo sich ein Performanceproblem verstecken kann? Leider nicht.

Die Werte, die Systemstatistiktools in Solaris auswerfen, haben ein großes Problem: Ihr bloßes Vorkommen ist vollkommen normal für den normalen Betrieb eines Betriebssystems.

Es gibt nur sehr wenige Werte, die in die in den meisten Fällen "Es läuft etwas schief" aussagen: Das klassische Beispiel ist hierbei zum Beispiel die „Scan Rate“ des virtuellen Speichersystems die vom Tool `vmstat` ausgegeben wird. Wenn hier mehr als 0 steht, ist der Speicher zu knapp, wobei sich der Administrator oft dessen schon bewusst ist, und „Scan Rate groesser 0“ dem Administrator lediglich sagt, das es nun wirklich Zeit für eine Speichererweiterung ist. Nur leider sind die Regeln nicht immer so einfach.

Systemcalls werden immer im Rahmen eines Programms ausgeführt. Auch Page Faults sind normal, Interrupts oder „spinning mutex locks“ können auch bei einem hochperformant laufenden Programm auftreten. Es ist die Menge, die entscheidend ist. Aber auch hier kann man nicht einfach die Regel „Viel ist schlecht!“ postulieren. Sämtliche Anzeigen der statistischen Tools sind Ausdruck eines laufenden Systems. Man muss ein Gefühl dafür entwickeln, was ein System macht, wenn es ordnungsgemäß läuft.

Aber was ist schon normalerweise normal?
Deswegen gilt die erste Grundregel der Leistungsdiagnose "Wisse immer, was normal für Dein System ist!". Man sollte also schon lange vor dem ersten Leistungsproblem damit anfangen, den Ist-Zustand des Systems zu erfassen. Man muss sich damit eine Baseline, eine Grundlage, schaffen mit der man einen fehlerhaften Zustand vergleichen kann.

Denn hat man erstmal das Leistungsproblem, so ist alles was man mit den Werten sieht, der Zustand des Systems zum Zeitpunkt des Fehlers und jede Mutmaßung, welcher Wert nun auf einen Fehler hinweisst, ist entweder pure Spekulation oder das Produkt von sehr viel Erfahrung. Nur der Vergleich mit der Baseline kann verraten, was sich geändert hat, kann eine Andeutung liefern, wo man suchen muss.

Die Erfassung dieser Baseline kann vielfältig erfolgen. Im einfachsten Fall dadurch, das man die Ausgabe von statistischen Werkzeugen einfach in eine Datei schreibt und diese gut weglegt. Aber auch das Erfassen in einer Datenbank ist möglich. Auf ein mögliches Tool zu diesem Zweck möchte ich am Ende des Artikels zu sprechen kommen.

Vom Hundertstel ins Tausendstel!
Am Ende des Suchens hat man möglicherweise eine ganze Menge Punkte gefunden, die Optimierungspotential bergen. Wo fängt man an? Die Antwort ist einfach: Das was am meisten Einsparungspotential beherbergt. Das klingt profan, wird aber oft übersehen. Denn man darf nicht von der absoluten Zeit des einzelnen Vorkommens ausgehen, sondern muss dies immer in Einheit mit der Häufigkeit des Vorkommens sehen.

Wenn ich etwa bei einer Transaktion 10 Sekunden von 11 Sekunden einsparen kann, klingt dies sicherlich beeindruckend. Kommt diese Transaktion allerdings nur alle zwei bis drei Stunden mal vor, ist die Gesamtersparnis gering. Schaffe ich es von einer Transaktion 0.1 Sekunden von 1 Sekunde

einzusparen, ist das zwar prozentual und absolut eher wenig. Ist das aber nun ein nahezu ständig vorkommender Teil der Applikation, so kann die Gesamtersparnis gewaltig sein.

Amdahl's Law – You can't run, you can't hide!
Amdahl's Law gehört zum großen Theoriegebäude der Informatik. Nur wird dessen Auswirkung auf die tägliche Praxis gerne übersehen. Wenn man nur eine Sache daraus mitnehmen möchte, so sollte man sich immer populärwissenschaftliche Variante von vergegenwärtigen: Die maximale Beschleunigung die durch Parallelisierung zu erwarten ist, ist vom seriellen Anteil abhängig.

Und der seriellen Anteile gibt es in einem komplexen System viele: Ein Netzwerkkabel beispielsweise, das der Kommunikation zwischen zwei Systemen dient. An einer bestimmten Position kann zu einer bestimmten Zeit nur ein Datenelement sein. Alles andere nennen wir bei Ethernet beispielsweise Kollision und führt zum Verwerfen beider Datenelemente.[2]

Das heißt, das es egal ist, mit welcher Parallelität eine Aufgabe auf einem Server ausgeführt wird, wenn beispielsweise ein Ethernet- oder FibreChannel-Netzwerk sich zwischen Komponenten befindet, wird hier der gesamte Ablauf serialisiert und wird somit möglicherweise zum Bottleneck.

Hier hilft oft der Einsatz schnellerer Netzwerktechnologien. So hat beispielsweise jeweils der Schritt von 10 Mbit/s nach 100 Mbit/s nach 1000 Mbit/s nach 10 Gbit/s bei Ethernet nicht nur den erwünschten Effekt einer erhöhten Übertragungskapazität gebracht. Mit jedem Schritt wurden auch die Latenzen im Netzwerk reduziert und somit auch der Punkt, an dem der Serialierer „Netzwerkkabel“ zum Bottleneck wird herausgeschoben. Sollte man also noch ältere Netzwerkstandards einsetzen, so lohnt sich zur Reduzierung von Latenzen der Blick auf neuere Standards.

Äußere Faktoren sind nicht zu unterschätzen
Wir sind eben schon auf äußere Faktoren zu sprechen gekommen, diese sind auf jeden Fall ein wesentlicher Bestandteil der Performance, die auf jeden Fall der Betrachtung bedürfen. Oftmals wird an bestimmten Faktoren optimiert, das Umfeld des Systems aber völlig vergessen.

Einige Beispiele:

- Bei näherer Beobachtung stellt sich heraus, das diese Transaktion aus 20 Datenbankabfragen auf einen anderen Server besteht. Der geneigte Beobachter mag bei Performanceproblemen zunächst einmal auf den Datenbankserver und auf den Applikationsserver schauen. Vergessen wird oftmals aber der Effekt des Netzes dazwischen: Nehmen wir mal ein Netzwerk auf Gigabit-Ethernet-Basis an. Gehen wir davon aus, das die Round-Trip-Time 0.2ms für einen einfachen Ping benötigt. Alleine das Hin- und Zurücklaufen des Paketes gibt vor, das in einer einzelnen Session nicht mehr als 5000 Datenbankabfragen pro Sekunde laufen können, sprich im oben Beispiel sind nicht mehr als 250 Userinteraktionen pro Sekunde möglich.

  Noch problematischer wird dies, wenn zwischen zwei System das Internet als verbindendes Element genutzt wird. Eine solche Verbindung erreicht leicht Latenzen im zwei- bis dreistelligen Millisekundenbereich. Den Effekt auf die maximale Anzahl von Transaktionen pro Sekunden kann man sich leicht ausrechnen.

- Ein Zugriff auf eine Festplatte ist immer ein sehr langwieriges Ereignis. Eine Festplatte liefert in etwa maximal 125 I/O-Operationen. Das heißt, selbst wenn die Platte sofort zur Verfügung steht, so braucht es mindestens 8 Millisekunden bis Daten geliefert werden. So lange man also nicht die Festplattenzugriffe durch Caches (sei es durch arbeitsspeicherbasierte Caches oder

2 Glücklichweise ist Kollision heute in Ethernetnetzwerken kein Thema mehr.

durch SSD) auf ein absolut nötiges Minimum reduziert hat, ist eine Suche nach Mikrosekunden oder Nanosekunden in Locks eigentlich nachrangig.

Meet the `*stats`

Solaris verfügt über eine große Anzahl von Tools aus der `*stat`-Reihe. Ich möchte hier nur auf eine kleine Auswahl zu sprechen kommen. Die `*stat`-Tools bieten einen sehr guten Überblick in den Zustand des Systems.

Beispielsweise kann mit `vmstat` festgestellt werden, wie sich das „virtual Memory"-Subsystem verhält.

```
jmoekamp@hivemind:~$ vmstat 1
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd cd cd cd   in   sy   cs us sy id
 0 0 51 882244 233300 35 62  0  1  1  0 36  5  3  5  3 2260 5736 1933  2  3 95
 1 0 58 1063484 311612 9 57  0  0  0  0  0  0  0  0  0 2068 5549 1903  1  3 97
 1 0 58 1063404 311632 4  7  0  0  0  0  0  0  0  0  0 1932 5373 1807  1  2 97
```

Mit `mpstat` lässt sich für jeden Prozessor ersehen, was ob dieser momentan mit Applikationen, mit dem Betriebssystem oder auch gar nicht (idle) beschäftigt ist. Auch werden eine Reihe von Events gezählt, die für die Abschätzung dessen, was auf den Prozessoren momentan läuft unabdingbar sind.

```
jmoekamp@hivemind:~$ mpstat 1
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0   21   0   68   705  155  385   24    1    1    0  1410    3   3   0  95
  1   14   0   56   514   95  556    9   15    6    0  1474    2   2   0  96
  2   13   0   64   494  128  476    7   14    5    0  1336    2   2   0  96
  3   14   0   49   547  249  517    8   13    6    0  1516    2   3   0  95
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0    0   525  100    0    0    0    1    0     0    0   1   0  99
  1    0   0   67   580   92  938    7   14    7    0  1296    1   4   0  95
  2    7   0    9   500   92  347    0   17    0    0   627    0   1   0  99
  3    0   0    0   648  301  528    3   13   10    0  1158    1   4   0  95
```

Für das I/O-Subsystem existiert mit `iostat` ein sehr ähnliches Tool:

```
jmoekamp@hivemind:~$ iostat 1
   tty       cmdk0         cmdk1         cmdk2         cmdk3           cpu
 tin tout kps tps serv  kps tps serv  kps tps serv  kps tps serv   us sy wt id
   0    1 396   5   24   77   3   15  391   5   24   77   3   14    2  3  0 95
   0  236   0   0    0    0   0    0    0   0    0    0   0    0    1  3  0 96
   0   80   0   0    0    0   0    0    0   0    0    0   0    0    1  2  0 96
```

Ein Tool liegt mir sehr am Herzen. Oftmals installieren Benutzer das von Linux bekannte top auf einem Solaris-System. Oftmals liegt das daran, das das Tool `prstat` den Nutzern nicht bekannt ist. Schon in seinem normalen Modus vermag das Tool `prstat top` vollständig zu ersetzen.

Ich möchte hier insbesondere auf den Betriebsmodus „Microstate Accounting" hinweisen. In diesem Modus wird sehr genau darüber Buch geführt,[3] wie lange ein Prozess in einem bestimmten Zustand (beispielsweise „auf ein Lock wartend" oder „auf eine CPU wartend") war und wie häufig bestimmte Ereignisse wie ein Systemcall oder ein „context switch" stattgefunden hat.

```
# prstat -m 1
PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/NLWP
  1097 jmoekamp 0,1 0,3 0,0 0,0 0,0  65  35 0,1 156   7  1K  29 VBoxHeadless/17
  8035 jmoekamp 0,1 0,2 0,0 0,0 0,0 0,0 100 0,0  21   0 279   0 gnome-netsta/2
 11579 jmoekamp 0,0 0,1 0,0 0,0 0,0 0,0 100 0,0  22   1 406   0 prstat/1
```

3 Anstatt – vereinfacht gesagt - nur jeweils zu bestimmten Momenten nachzugucken, in welchem Zustand ein Prozess ist und das für den gesamten Zeitintervall anzunehmen.

```
  8043 root     0,0 0,1 0,0 0,0 0,0 0,0 100 0,0  70   0 100   0 gnome-netsta/1
  3418 root     0,1 0,0 0,0 0,0 0,0  33  67 0,0  14   3 152   0 Xorg/3
    49 root     0,0 0,0 0,0 0,0 0,0 0,0 100 0,0   6   1  74   3 in.mpathd/1
  9516 jmoekamp 0,0 0,0 0,0 0,0 0,0  67  33 0,0 141   2 643   0 soffice.bin/15
  7990 jmoekamp 0,0 0,0 0,0 0,0 0,0 0,0 100 0,0  12   0  50   0 nautilus/1
  7967 jmoekamp 0,0 0,0 0,0 0,0 0,0 0,0 100 0,0  12   0  50   0 gnome-panel/1
  8033 jmoekamp 0,0 0,0 0,0 0,0 0,0 0,0 100 0,0  13   0  55   0 clock-applet/1
  9492 jmoekamp 0,0 0,0 0,0 0,0 0,0 0,0 100 0,0  12   0  50   0 file-roller/1
  8022 jmoekamp 0,0 0,0 0,0 0,0 0,0 0,0 100 0,0  12   0  31   0 gvfsd-trash/1
  9488 jmoekamp 0,0 0,0 0,0 0,0 0,0 0,0 100 0,0  13   0  34   0 trackerd/1
  8010 jmoekamp 0,0 0,0 0,0 0,0 0,0 0,0 100 0,0  12   0  31   0 gvfs-hal-vol/1
   684 postgres 0,0 0,0 0,0 0,0 0,0 0,0 100 0,0  25   0  75   0 postgres/1
  9562 jmoekamp 0,0 0,0 0,0 0,0 0,0  50  50 0,0  13   0  55   0 evince/2
  7957 jmoekamp 0,0 0,0 0,0 0,0 0,0 0,0 100 0,0  13   0  55   0 gnome-settin/2
  1106 jmoekamp 0,0 0,0 0,0 0,0 0,0  89  11 0,0  21   0  52   0 VBoxSVC/9
   685 postgres 0,0 0,0 0,0 0,0 0,0 0,0 100 0,0  25   0  50   0 postgres/1
     5 root     0,0 0,0 0,0 0,0 0,0 0,0 100 0,0 175   1   0   0 zpool-rpool/41
   645 root     0,0 0,0 0,0 0,0 0,0 0,0 100 0,0  11   0  11   0 squid/1
Total: 133 processes, 620 lwps, load averages: 0,13, 0,13, 0,12
```

Mit der Beschreibung der Bedeutung einer jeden Zeile könnte man bereits ein längeres Buch füllen. Zu mindestens auf einem Solaris-System ist aber jeder dieser Befehle mit einer sehr guten Manual Page hinterlegt, die die Bedeutung der Zahlen gut erklärt, auch wenn hier das schon erwähnte Grundwissen zum Thema Unix nicht fehlen sollte..

Allgemein gesprochen geben alle diese Tools bereits eine gute Übersicht über den Zustand des Systems. Diese Zahlen können einen ersten Hinweis geben, ob etwas in einem System etwas nicht in Ordnung ist.

Allerdings geht dies wie schon erläutert nur über Mutmaßungen hinaus, wenn man auch weiß, wie diese Werte normalerweise aussehen, wenn alles in Ordnung sind. Beispielsweise können 1400 Systemcalls auf einer CPU völlig normal für eine Applikation sein, sie können genauso auf ein Problem hinweisen. Wichtig ist hier immer der Vergleich.

DTrace braucht man am Anfang nicht, zu mindestens eher sehr selten ...
Oftmals fangen Präsentationen über Performance, die auf die besonderen Vorzüge von Solaris 10 hinweisen sollen, gleich mit DTrace an. Warum fange ich nicht gleich mit DTrace an, sondern weise zunächst auf einige herkömmliche Tools hin?

DTrace ist keine Silberkugel zum Finden von Performanceprobleme. DTrace beantwortet Fragen, es ist nicht so, das DTrace Frage und Antwort gleichzeitig liefert..

DTrace ist ein hervorragendes Tool um herauszufinden, was das System genau treibt. Der Nachteil: Es gibt in einem Solaris System über hunderttausende Messpunkte Es ist die Kunst, an das DTrace-Subsystem die richtigen Fragen zu stellen. Um diese Fragen finden und stellen zu können, sind die Tools der *stat-Serie das Mittel der Wahl.

Ein Beispiel: Mit `mpstat` fällt mir auf, das die Prozessoren sehr viele Systemcalls verarbeiten und die Zeit, die das System im Betriebssystemkernel verbringt, relativ hoch ist. Ich kann also die Fragen formulieren „Welche Systemcalls werden ausgeführt?“ „Wie lange benötigt ein Systemcall zur Ausführung?“ oder „Welche Systemcalls verbrauchen die meiste Zeit bei der Ausführung des Programms?“.

DTrace steht selten am Anfang einer Optimierung, einer Bottleneck-Suche. DTrace ist immer die Instrument der Überprüfung einer Theorie, warum ein System langsam ist, es ist immer ein Instrument, um sich einen bestimmten Bereich sehr gezielt anzugucken. Worauf man das Instrument

richtet, sollte man aber schon vorher wissen und die genannten `*stat`-Tools sind dazu eine große Hilfe.

Die Mächtigkeit von DTrace

Ich möchte aber doch einmal auf dieses Tool „DTrace“ zu sprechen kommen. DTrace als Tool hilft bei der Suche nach Performance-Problemen ungemein, da es Fragen beantworten kann, die sich früher einer einfachen Beantwortung entzogen haben.

Um dies zur ermöglichen, wurden in Solaris 10 eine Vielzahl von Messpunkten eingebaut, die während des Betriebes ein- und wieder ausgeschaltet werden können. Die Zahl dieser Messpunkte geht mittlerweile in die Zehntausende. Die Messpunkte sind ständig verfügbar, sie haben keine Auswirkungen auf die Performance, wenn sie nicht benutzt werden und auch eingeschaltet, ist die Auswirkung eines einzelnen Messpunktes minimal. Außerdem ist DTrace darauf ausgelegt, die Stabilität eines System niemals zu gefährden.[4]

Ich möchte hier zum Beispiel den Desktoprechner, auf dem dieses Dokument entstanden ist, als Beispiel nehmen:

```
# dtrace -l | wc -l
90399
```

Auf diesem System sind über 90.000 Messpunkte verfügbar, die mir einen genauen Blick in die Abläufe des Systems bereitstellen.

Doch welcher Tiefe sind diese Daten? Ich möchte hier mit einem Beispiel aus den Abgründen des Prozess-Scheduling-Systems beginnen: In einem Timesharing-System werden Prozesse von einer CPU genommen sobald diese entweder nichts mehr zu rechnen haben oder ein bestimmtes Quantum Zeit aufgebraucht ist. Doch was macht das System zu genau diesem Zeitpunkt, wenn die CPU freigeräumt wird. Mit DTrace lässt sich diese Frage einfach beantworten:

```
# dtrace -n \'sched:::off-cpu /execname == \"SomeApp\"/{@[ustack()] = count()}END{trunc(@,5)}\'
dtrace: description \'sched:::off-cpu \' matched 3 probes
^C
CPU ID FUNCTION:NAME
1 2 :END
libc.so.1`___lwp_cond_wait+0x4
libjvm.so`__1cCosNPlatformEventEpark6M_v_+0x100
libjvm.so`__1cHMonitorFIWait6MpnGThread_x_i_+0xdc
libjvm.so`__1cHMonitorEwait6Mblb_b_+0x350
libjvm.so`__1cKGangWorkerEloop6M_v_+0xc8
libjvm.so`java_start+0x22c
libc.so.1`_lwp_start
553
[...]
```

Ich kann hiermit feststellen, das die meisten Herunternahmen von einer CPU stattfanden, während der Thread auf etwas gewartet hat, es sich somit um einen voluntary context switch handelt und nicht um einen involuntary context switch, der einen Thread von einer CPU zwingt, auch wenn dieser eigentlich noch rechnen könnte.

Im Gegensatz zu anderen statistischen Tools lässt sich also nicht nur sagen ob es sich um einen voluntary oder involuntary Context Switch handelt, sondern auch, wo gerade das Programm sich in

4 Wobei man natürlich ein System extrem verlangsamen kann, wenn man alle Messpunkte gleichzeitig einschaltet, wie es dem Autor auf einem System einmal passiert ist.

seiner Ausführung befand, als es vom Scheduler des Betriebssystems von der CPU genommen worden ist.

Eine andere sehr interessante Verwendung der DTrace-Funktionalität nutzt die Tatsache aus, das DTrace nicht nur eine Anzahl von Messpunkten im Betriebssystem ist, sondern gleichzeitig auch eine Programmiersprache bereitstellt, um Auswertungen durchzuführen. Beispielsweise lässt sich sehr leicht zählen, wie viele Systemcalls von einer Applikation ausgeführt werden:

```
jmoekamp@hivemind:~# dtrace -n 'syscall:::entry { @num[execname] = count(); }'
dtrace: description 'syscall:::entry ' matched 239 probes
^C

  nscd                                                              1
[...]
  thunderbird-bin                                               16752
  VBoxHeadless                                                  19819
```

Diese Zahl ließe sich noch einfach durch einen Blick auf das `prstat`-Kommando ermitteln, allerdings ist es mit DTrace ebenso einfach auch Statistiken zu erhalten, welcher Art diese Systemcalls sind.

```
jmoekamp@hivemind:~# dtrace -n 'syscall:::entry { @num[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 239 probes
^C

  access                                                            1
  fcntl                                                             1
[...]
  read                                                          11593
  llseek                                                        11964
  ioctl                                                         25703
```

So lässt sich schnell genau ermitteln, in welcher Weise das Betriebssystem im laufenden Betrieb genutzt wird.

In der Kürze der Zeit lässt sich natürlich nicht umfassend auf die Fähigkeiten von DTrace eingehen, ich möchte hier zum Beispiel auf das Buch „Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris“ von Richard McDougall, Jim Mauro, Brendan Gregg hinweisen, das dieses Thema sehr gut behandelt.[5]

Use Cases

Bleibe bei den Defaults, aber traue ihnen nicht blindlings
Wichtig beim Tuning von System ist, das man sich so lange an die Defaults halten sollte, wie nur möglich. Erstens sind diese Defaults oft Ergebnis von Erfahrung und Tests. Da technische Systeme immer eine gewisse Komplexität besitzen, hat jede Änderung wieder Auswirkungen auf andere Teilkomponenten. Zweitens ist jedes Tuning ein administrativer Eingriff, der womöglich irgendwann wieder vergessen wird, und bei anderen Softwareversionen vielleicht sogar kontraproduktiv ist.

Ich möchte hier ein Beispiel zeigen, das der Praxis entstammt. Ein Kunde hat eine Java-Applikation in Betrieb. Der Kunde hat zum Betrieb ein System mit 24 CPUs in 7 Solaris Zonen aufgeteilt und betreibt in jeder Zone drei dieser Java-Prozesse. Das System läuft in dieser Form sehr lange sehr stabil. Nach einem Softwareupdate kommt es zu einem Systemausfall. Merkwürdigerweise

5 Richard McDougall, Jim Mauro, Brendan Gregg - „Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris“ - http://amzn.to/cvW0jW – ISBN 0131568191

verschwindet das Problem, sobald man sogenannte Prozessorsets einrichtet, die jeden Java-Prozess auf eine Untergruppe von maximal 4 CPU limitieren.

Was war passiert? Es handelte sich hierbei um einen wenig bekannten Mechanismus in Java. Dieser tritt in Kraft wenn man mit sogenannter Parallel Garbage Collection arbeitet. Java errechnet dann aus der Menge der zur Verfügung stehenden Prozessoren die Anzahl der parallel laufenden Garbage Collection Prozesse. Dieser Mechanismus führt in den meisten Fällen zu guten Ergebnissen. Das Problem – und damit sind wir bei dem Punkt „Traue den Defaults nicht blindlings“ - war in diesem Fall, das die automatische Festlegung der Defaults für das System von falschen Annahmen ausgegangen ist.

Aber von Anfang an: Nach einem Konfigurationsfehler war ein Prozessorset[6] plötzlich doppelt so groß, wie eigentlich vorgesehen. Die Reaktionszeit der Prozessoren auf dem Prozessorset verschlechterte sich schlagartig. Um überhaupt erst einmal einen Überblick über die Probleme zu kommen, wurde als erstes das mpstat Tool genutzt.

```
# mpstat
CPU minf mjf xcal intr ithr csw  icsw migr smtx srw syscl usr sys wt idl
96  2    0   471  563  6    1549  562 277    77 0    8377 94    6  0 0
97  3    0   397  570  6    1417  591 294    67 0    9141 94    6  0 0
98  3    0   315  496  6    1681  554 313    60 0    8794 94    6  0 0
99  2    0   451  704  6    1915  706 314    76 0   10715 93    7  0 0
100 3    0   464  563  5    1760  678 312   195 0    7451 94    6  0 0
102 5    0   352  494  5    1585  533 290    45 0    8357 94    6  0 0
103 7    0   392  570  5    1633  623 312    40 0   10005 94    6  0 0
```

Die hohe Anzahl der Systemcalls war sehr auffällig, auch die Tatsache, das das System unter Vollast stand (0% Idle) war merkwürdig. Damit hatten wir eine erste Fragestellung für DTrace „Was sind das für Systemcalls, und wer führt diese aus?“

```
# dtrace -n 'syscall:::entry { @num[ustack()] = count(); }'
  libc.so.1`lwp_yield+0x4
 libjvm.so`bool ParallelTaskTerminator::offer_termination(TerminatorT erminator*)+0x90
  libjvm.so`void ParEvacuateFollowersClosure::do_void()+0x9c8
  libjvm.so`void ParNewGenTask::work(int)+0x148
  libjvm.so`void GangWorker::loop()+0x84
  libjvm.so`java_start+0x22c
  libc.so.1`_lwp_start
24859

 libc.so.1`lwp_yield+0x4
 libjvm.so`bool ParallelTaskTerminator::offer_termination(TerminatorT erminator*)+0x90
 libjvm.so`void ParEvacuateFollowersClosure::do_void()+0x9c8
 libjvm.so`void ParNewRefProcTaskProxy::work(int)+0xf0
 libjvm.so`void GangWorker::loop()+0x84
 libjvm.so`java_start+0x22c libc.so.1`_lwp_start
30074
```

Mit dieser Ausgabe beantwortete DTrace eben diese Fragen. Der häufigste Systemcall war lwp_yield, und überwiegend wurde dieser von einer Funktion mit dem Namen `ParallelTaskTerminator::offer_termination` aufgerufen. Guckt man im Source des JDK6 nach, so findet man auch genau diesen `yield()`-Aufruf.

```
jmoekamp@hivemind:/datapool/bibliothek/sources/jdk6/hotspot# grep -R
"ParallelTaskTerminator::offer_termination" *
src/share/vm/utilities/taskqueue.cpp:
ParallelTaskTerminator::offer_termination(TerminatorTerminator* terminator)
```

6 Prozessorsets sind in Solaris - vereinfacht gesagt – ein Feature, mit der Prozessen nur eine Untermenge der verfügbaren CPUs gezeigt werden, egal wie viele CPUs sich sonst noch im System befinden.

```
src/share/vm/utilities/taskqueue.cpp: gclog_or_tty-
>print_cr("ParallelTaskTerminator::offer_termination() "
```

Um kurz in den Source zu gucken:

```
114         if (hard_spin_count > WorkStealingSpinToYieldRatio) {
115           yield();
116           hard_spin_count = 0;
117           hard_spin_limit = hard_spin_start;
118 #ifdef TRACESPINNING
119           _total_yields++;
120 #endif
121         } else {
122           // Hard spin this time
123           // Increase the hard spinning period but only up to a limit.
124           hard_spin_limit = MIN2(2*hard_spin_limit,
125                                  (uint) WorkStealingHardSpins);
126           for (uint j = 0; j < hard_spin_limit; j++) {
127             SpinPause();
128           }
```

Sucht man dann noch nach `ParNewGenTask::work(int)`, so kann der Bereich eingegrenzt werden werden, in dem die mögliche Quelle des Problems ist.

```
jmoekamp@hivemind:/datapool/bibliothek/sources/jdk6/hotspot# grep -R "ParNewGenTask::work" *
src/share/vm/gc_implementation/parNew/parNewGeneration.cpp:void ParNewGenTask::work(int i) {
```

Damit war klar, das es sich um ein Problem in der Garbage Collection handelt. Nun ergab sich die Frage, welche Threads so häufig diesen yield call auslösten. Diese konnte wiederum mit DTrace beantwortet werden[7]:

```
root@server:~ dtrace -n 'syscall::yield:entry {@[pid, tid] = count()}'
dtrace: description 'syscall::yield:entry ' matched 1 probe
[...]
14344  4                 25000
14344  3           26000
14344  7                 27000
14344  5           28000
14344  6                 29000
14344  2           30000
14344  8                 31000
[...]
^C
```

pid steht hierbei für Process ID und tid steht für Thread ID. Sieben Threads. An dieser Stelle klingelte eine Alarmglocke, da diese Zahl schon einmal auftauchte. Die Zone lief in einem Prozessorset mit sieben CPUs.

Nach einer etwas ausgedehnten Suche (5 Minuten … via Google) konnte dann herausgefunden werden, was das eigentliche Problem ist: Java 1.6 leitet die Anzahl der Garbage Collection Threads von der Anzahl der CPUs ab. Das Regelwerk ist sehr einfach: Bis zu 8 Prozessoren ist die Anzahl der Garbage Collection Threads gleich der CPU-Anzahl, darüber hinaus sind es 8 plus fünf Achtel der CPUs über 8 CPUs.

An sich ist das eine sehr gute Defaulteinstellung. Allerdings weiß ein Java-Prozess nichts davon, das da möglicherweise auch noch andere Java-Prozesse sind, die auch Garbage Collection Prozesse starten und somit hält sich jede JVM an ihre Defaultregel. Macht man das mit 6 JVM in einer 7 CPU Zone

7 Die Zahlen entsprechen nicht der echten Ausgabe, da diese nicht mehr vorliegen, stellen aber eine relativ genaue Repräsentation dar.

laufen da plötzlich 42 Garbage-Collection, die sehr schnell hintereinander sehr viele yield-Calls durchführen. Ein einfaches `-XX:ParallelGCThreads=3` und `-XX:ParallelCMSThreads=1` löste das Problem.

Dieser Fall ist ein gutes Beispiel gewesen, das Defaults meistens ein sehr gutes Ergebnis liefern, die genaue Herkunft dieser Defaults aber hinterfragt werden sollte, damit man weiß, das man eine der Grundannahmen, auf denen diese Standardeinstellung basiert, verletzt hat und darüber ein Performanceproblem schafft.

Interessanterweise müssen diese nicht gleich auffallen. Oftmals sind Systeme so leistungsfähig, das das Problem nicht auffällt. Nach einem Releasewechsel kann sich aber die Nutzungscharakteristik der Applikation ändern. Manchmal muss die Änderung nur minimal sein, um dann ein Bottleneck zu verursachen, der die gesamte Applikation in Mitleidenschaft zieht.

Leicht verdiente Performance

Ich habe zwar geschrieben, das man eine Baseline benötigt um vernünftig auf der die Suche nach Performanceproblemen gehen zu können und das DTrace selten am Anfang der Suche steht, aber einige Ausnahmen gibt es schon.

Eine dieser Ausnahmen ist plockstat. plockstat basiert DTrace und ist in der Lage, bei einem laufenden Prozess zu ermitteln wie lange dieser durch Locks aufgehalten wird.

Im Rahmen eines Kundentermins wurde also dieser Befehl abgesetzt, um überhaupt erstmal zu Prüfen, ob es sich um ein Locking-Problem handeln könnte.

```
# plockstat -A -p 18107:
      Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
78    69427 libc.so.1`libc_malloc_lock   libzip.so`Java_java_util_zip_Deflater_deflateBytes+0x29c
44    99867 libc.so.1`libc_malloc_lock   libzip.so`Java_java_util_zip_Deflater_deflateBytes+0x36c
21    35373 libc.so.1`libc_malloc_lock   libzip.so`Java_java_util_zip_Deflater_deflateBytes+0x29c
11    58851 libc.so.1`libc_malloc_lock   libzip.so`Java_java_util_zip_Deflater_deflateBytes+0x2dc
9    71014 libc.so.1`libc_malloc_lock   libzip.so`Java_java_util_zip_Deflater_deflateBytes+0x36c
[... ca 25 Zeilen deleted ...]
1    10483 libc.so.1`libc_malloc_lock   libnsl.so.1`netconfig_free+0x10
1     4401 libc.so.1`libc_malloc_lock   libnsl.so.1`netconfig_free+0x
```

Hierbei wurde `libc_malloc_lock` in durchaus reichlicher Anzahl vorgefunden. Dieser Lock ist der zentrale Lock des Memory Allocators der in der libc enthalten ist. Dieser ist sozusagen der Standardallocator in Unix und somit auch in Solaris. Er hat nur einen erheblichen Nachteil. Eben dieses zentrale Lock, wenn eine Vielzahl von Threads Speicher allozieren oder deallozieren möchten, so müssen sie erst dieses Lock erlangen, bevor sie weiterarbeiten können. Das kann die Ausführung erheblich verlangsamen. Im obigen Beispiel summierten sich die Wartezeiten auf 15126580ns in 10 Sekunden. Aber beim gleichen Kunden fiel eine zugekaufte Applikation auf, die von 10 Sekunden 1.7 in eben diesem `libc_malloc_lock` stand.

Allerdings war sich Sun dieses Problems früh bewusst, und hat deshalb in Solaris 8 und 9 mit `libumem` und `mtmalloc` zwei Memory Allocator Bibliotheken bereitgestellt, die damit umgehen können, das viele Threads gleichzeitig auf Speicher zugreifen. Man kann diese Bibliotheken einer Applikation recht einfach unterschieben.

Dazu gibt es die Umgebungvariable LD_PRELOAD. Dadurch wird vereinfacht gesagt sichergestellt, das zunächst Funktionen aus den mit dieser Umgebungsvariablen spezifizierten Bibliotheken genutzt werden, bevor beispielsweise jene aus libc genutzt werden. Da wir die malloc-Funktion aus der libc

nicht mehr verwenden wollen, stattdessen aber jene aus der libumem zum Tragen kommen soll, können wir genau diese Umgebungsvariable verwenden.
Dementsprechend muss man also nur `LD_PRELOAD=libumem ./someapp` angeben, um der Applikation die neue Library unterzuschieben. Nach einem Neustart der Applikation sah das Ergebnis dann gleich sehr viel erfreulicher aus:

```
# plockstat -A -p 18808
Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
    5    37929 0x3b400                      libumem.so.1`umem_cache_free+0x6c
    2    76420 0x3b400                      libumem.so.1`umem_cache_alloc+0x50
   12     5828 0x3b400                      libumem.so.1`umem_cache_free+0x6c
    2    31008 libumem.so.1`vmem0+0x4a8     libumem.so.1`vmem_alloc+0xf0
    8     6141 0x3b400                      libumem.so.1`umem_cache_alloc+0x50
    1    40411 libumem.so.1`vmem0+0x4a8     libumem.so.1`vmem_xalloc+0x140
    1     5922 0x37080                      libumem.so.1`umem_cache_alloc+0x50
```

Insgesamt summierte sich die Zeit in diesem Lock auf gerade einmal 569898ns. Das sind 3.768% Prozent der Zeit, die die gleiche Applikation im im libc_malloc_lock wartend verbracht hat. Bei der genannten Applikation die 1.7 von 10 Sekunden auf libc_malloc_lock gewartet hat, war die Änderung des Memory Allocators genauso effektiv.

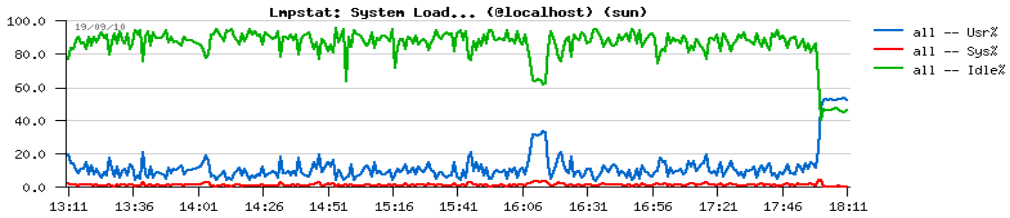
Hier kann man also wirklich eine einfache Regel definieren: Hat man eine Applikationsumgebung mit vielen Threads, so lohnt es sich mit plockstat nachzusehen, wie lange diese Applikation in Locks wartet. Ist `libc.so.1`libc_malloc_lock` unter diesen, so lohnt es sich immer, einmal libumem auszuprobieren, um zusätzliche Performance zu erlangen. Einfacher und mit weniger Aufwand erhält man einen Performanceschub selten.

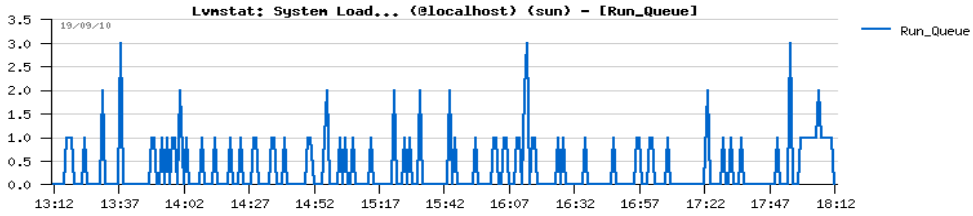
dim_STAT – ein kostenloses Tool zum Monitoring von Unix-Systemen
In der täglichen Praxis habe ich schon häufiger festgestellt, das das von einer guten Graphik unterstützte Auge das schnellste Mittel ist um Muster oder Unregelmäßigkeiten in Lastverläufen zu finden. In langen Zahlenreihen wird man selten etwas schnell entdecken. Daher ist es sehr hilfreich, wenn man ein Tool nutzt, das zum einen Daten sammeln kann zum anderen diese aber auch sinnvoll graphisch darstellen kann. Daher bieten viele Monitoring Toolkit auch genau solche Funktionen.

Will man die Zeit bis zur Einführung eines richtigen Performance Monitoring Frameworks überbrücken[8] beziehungsweise sich die Einführung eines kostenpflichtigen Tools sparen, so hat sich in meiner Praxis die Nutzung von dim_STAT als sehr gutes und ad-hoc verfügbares Werkzeug herausgestellt. Es handelt sich dabei um eine von einem Oracle-Mitarbeiter geschriebenes Tool zur Erfassung von Systemparametern, die durch zahlreiche im Betriebssystem verfügbare Werkzeuge zur Ausgabe von Nutzungsstatistiken geliefert werden, und dessen Umwandlung in graphische Übersichten. Anders als viele Tools werden die Rohdaten vollständig einer Datenbank vorgehalten und nicht ausgedünnt. So lassen sich genaue Analysen mit historischen Daten auch noch nach längerer Zeit durchführen.

Mit einfachen Mitteln lassen sich so Übersichten generieren. Als Beispiel habe ich hier einmal zwei sehr einfache Übersichten gewählt:

8 Oder sich die Diskussion mit dem Operating, ob die Aufzeichnung dieses Werts nun sinnvoll ist, sparen ...





Erhältlich ist dieses Tool zum Download unter http://dimitrik.free.fr/ .

Abschließende Bemerkungen

Wahrscheinlich haben Sie sich von diesem Text und dem darauf basierenden Vortrag ein Kochbuch zum Auffinden von Performanceproblemen erwartet. Ein solches Kochbuch kann es nicht geben, denn es gibt fast ebenso viele Gründe für Performanceprobleme, wie Performanceprobleme gibt. Ich hoffe aber, das Ihnen dieser Text einige Einsichten gegeben hat, die Ihren Blick für mögliche Problemstellen schärft.

Kontaktadresse:

Jörg Möllenkamp
Oracle B.V. & Co. KG
Nagelsweg 55
D-20098 Hamburg

Telefon: +49 (0) 12-345 6789
E-Mail joerg.moellenkamp@oracle.com
Internet (dienstlich): http://www.oracle.com
Internet (privat): http://www.c0t0d0s0.org