Teamwork - DTrace und Analytics

Thomas Nau
Universität Ulm – kiz
Ulm

Schlüsselworte
DTrace, Performance, Analyse, Monitoring, StatsStore, Analytics

Einleitung

Den Zustand eines Server-Systems zu jeder Zeit analysieren zu können, ist im heutigen Zeitalter von Virtualisierung und Systemen mit dutzenden von CPU-Kernen wichtiger denn je um potentielle Engpässe und Störungen pro-aktiv erkennen zu können.

Wechselwirkungen zwischen Anwendungen und deren Konkurrenz um Ressourcen, etwa im Speicher- und IO-Bereich, sind nur ein Beispiel das in der Praxis oft auftritt, jedoch schwer analysierbar ist. Dies gilt insbesondere dann, wenn die Störungen nur unregelmäßig auftreten oder von kurzer Dauer sind.

Eine ideale Lösung sollte leichtgewichtig sein, jedoch auch die Möglichkeit bieten auf historische Daten zurück greifen zu können um Trends zu erkennen.

Seit ihrem Erscheinen in Solaris 10 vor über einem Jahrzehnt setzt DTrace, die *dynamic tracing facility,* Maßstäbe im Bereich der System-Analyse und des Performance Monitoring und ist auch heute noch das Tool an dem sich die Werkzeuge anderer Betriebssysteme und Distributionen messen lassen müssen. Auf Linux Systemen hat der 2013 erstmals vorgeschlagene eBPF (*Enhanced Berkeley Packet Filter*) die Möglichkeit geschaffen mit einem Tool zu DTrace vergleichbare Ergebnisse zu erzielen. Allerdings muss auch erwähnt werden, dass die Lernphase für eBPF deutlich schwieriger ist und länger dauert als für DTrace.

Oracle baut DTrace im Solaris Umfeld aktiv als Grundlagen-Technologie weiter aus, setzt also andere Tools auf den DTrace Schnittstellen auf, und passt es an aktuelle Entwicklungen des Kernels an. Darüber hinaus gibt es immer wieder Gerüchte, Oracle plane eine vollständige Linux Implementierung. Ob und wann das der Fall sein wird bleibt eben so offen, wie die dann notwendige Akzeptanz auf Seiten der Linux-Administratoren zumal eBPF hinsichtlich der Funktionalität als durchaus gleichwertig betrachtet werden kann.

Ergänzt werden diese Möglichkeiten mit Solaris 11.4 nun durch den StatsStore und einem zugehörigen Web-Interface. Damit ist nun auch die Möglichkeit vorhanden, historische Daten mit aktuellen zu vergleichen und Trends zu erkennen.

Auf den folgenden Seiten werden die wesentlichen Neuerungen von DTrace vorgestellt sowie erste Eindrücke des Web-Interface und der zu Grunde liegenden Technologie vermittelt.

Hintergrund des Autors

Der Autor ist Leiter der Abteilung Infrastruktur des Kommunikations- und Informationszentrums (kiz) der Universität Ulm und gleichzeitig dessen stellvertretender Leiter. Das kiz trägt unter anderem die Gesamtverantwortung für die universitäre IT-Infrastruktur, inklusive Telefonie, sowie die Versorgung der Wissenschaftler und Studenten sowohl mit elektronischen als auch mit Print-Medien. Die

Kernaufgaben der Abteilung Infrastruktur umfassen hierbei insbesondere Planung, Weiterentwicklung und den Betrieb der Netzwerke, sowie aller zentralen Server. Zu diesen zählen neben Backup- und HPC-Systemen insbesondere auch die "virtuellen Welten" und die auf HA-Clustern basierenden Mail-, Datenbank- und File-Server der Universität Ulm. Nach wie vor ist Solaris die Grundlage von vielen dieser kritischen IT-Dienste, insbesondere derer die „storage-lastig“ sind.

Darüber hinaus ist die Abteilung sehr stark in Projekte[1] des Landes Baden-Württemberg eingebunden und erbringt in diesem Zusammenhang auch Dienstleistungen für weitere Hochschulen des Landes basierend auf dem leistungsfähigen Landeshochschulnetz BelWü[2].

Historie

Der Betrieb einer zentralen IT-Infrastruktur erfordert insbesondere auch die effiziente und effektive Erfassung und Auswertung von Performance- und System-Kennzahlen für das Monitoring aber auch zu Planungszwecken und zur Analyse im Problem- oder Fehlerfall. Derartige Tools sind daher seit jeher Bestandteil jedes zum Einsatz kommenden Betriebssystems bzw. dessen Distributionen. Eine Charakterisierung der zu Grunde liegenden Technik ist wie folgt:

DTrace know-how Auffrischung, Basics:

Durch seine dynamische Natur und Struktur sowie die Integration in den Kernel bietet DTrace in aller Regel eine Lösung mit Sicht auf das Gesamtsystem ohne den Einschränkungen anderer Tools, etwa denen die nur statistische Aussagen liefern oder sampling-basiert arbeiten, zu unterliegen. Im Kern besteht DTrace hierbei aus nur wenigen Komponenten, deren Nutzung oft auch ohne root-Privilegien möglich ist.

Probes sind im Kernel, seinen Modulen, Bibliotheken und Anwendungen verteilte Triggerpunkte, die dynamisch zur Laufzeit aktiviert und deaktiviert werden können, ein gravierender Unterschied zum sampling-Ansatz. Das Auslösen einer aktivierten *probe* kann vielfältige Operationen, etwa das Sammeln oder auch die Manipulation von Daten – in engen Grenzen – nach sich ziehen. In einem aktuellen Solaris 11.3 System stehen mehr als 110.000 *probes* zur Verfügung, deren größte Gruppe die Kernelfunktionen sind. Die Namens-Syntax zur Definition von *probes* folgt dem Schema

```
provider:module:function:name

sysinfo:genunix:pread64:readch
profile:::tick-5s
```

Funktionell werden diese *probes* in sogenannte *provider* gruppiert. Deren wichtigste Aufgabe ist die Bereitstellung der Daten in aufbereiteter Form. So stellt z.B. der *ip-provider* Adressen als String und nicht in binärer Form zur Verfügung. Äußerst hilfreich sind die *provider,* die die Aufbereitung von protokollspezifischen Daten übernehmen. Neu hinzugekommen sind im aktuellen 11.4 Release: *SCSI, ICMP, IGMP*, *SCTP* sowie *PCAP* und *FileOPs*. Dazu gleich mehr in den Besipielen

Unglücklicherweise ist der *SMB-provider* nicht im DTrace Manual, sondern im Solaris Handbuch[3] *Managing SMB File Sharing and Windows Interoperability* zu finden.

1 bwCloud: Standortübergreifende Servervirtualisierung
bw100G: Forschung und innovative Dienste für ein flexibles 100G-Netz in Baden-Württemberg
bwHPC, bwHPC-C5: http://www.bwhpc-c5.de

2 http://www.belwue.de

3 https://docs.oracle.com/cd/E37838_01/html/E61013/smbdtrace.html#scrolltoc

Ein weiterer wesentlicher Bestandteil der DTrace Architektur sind die sogenannten *consumer*. Sie sind für das asynchrone Auslesen der durch die *provider* bereitgestellten Daten aus den Puffern des Kernels verantwortlich. Die Schnittstelle zum Solaris Kernel ist wie üblich über Bibliotheken (*libdtrace)* und Gerätetreiber implementiert. Mit 11.4 besteht nun auch offiziell und dokumentiert die Möglichkeit eigene *consumer* zu entwickeln. An dieser Stelle sei jedoch lediglich auf die Oracle Dokumentation[4] verwiesen.

Zu guter Letzt bedient sich DTrace einer eigenen Sprache namens „D“, die an „C“ angelehnt ist und die meisten von dort bekannten Operatoren und Typen zur Verfügung stellt. Im Gegensatz zu üblichen Anwendungen hat ein D-Programm keinen Programmfluss, sondern seine Blöcke agieren als einzelne Routinen, die durch das Auslösen aktivierter *probes* aufgerufen werden.

Abbildung 1 verdeutlicht die Integration in den Solaris Kernel. Für weitergehende Informationen zu den Grundfunktionen sei an dieser Stelle lediglich auf die Artikel des Autors in den Tagungsbänden der *DOAG 2013, 2014, 2015* und *2017* sowie auf die Oracle Dokumentation unter

```
https://docs.oracle.com/cd/E37838_01/html/E61035/index.html
```

verwiesen.

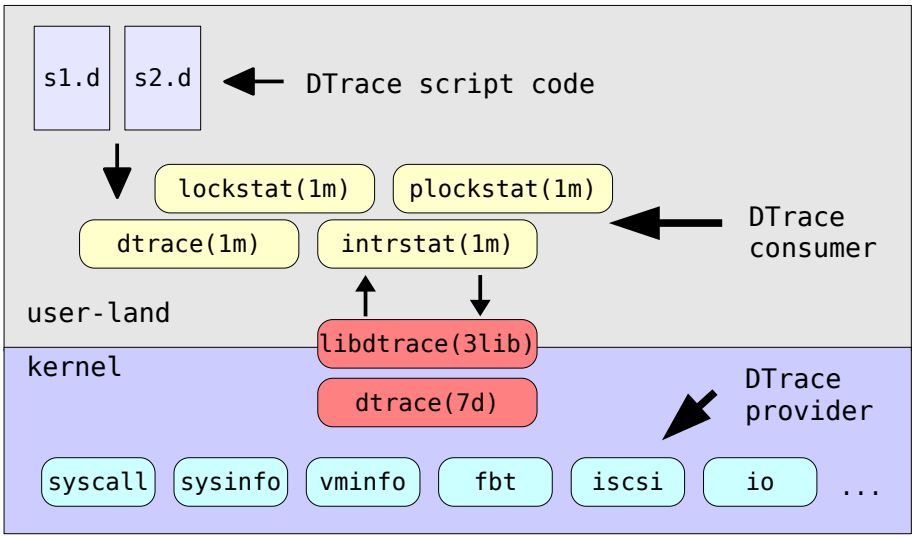

Abbildung 1: DTrace user-land / Kernel Schnittstelle

DTrace know-how Auffrischung, Aggregations, der DTrace Warp-Antrieb:
Neben den von DTrace bereits vordefinierten Variablen und Strukturen ist vor allem der Einsatz von *aggregations* bei komplexeren Fragestellungen unabdingbar. Allen ist hierbei die nachfolgende mathematische Eigenschaft gemein: Wird die Funktion im ersten Schritt auf Teilmengen der Daten angewandt und im zweiten Schritt auf diese Zwischenergebnisse so liefert dies identische Ergebnisse wie die Anwendung auf die Gesamtmenge aller Daten in einem Zug. Beispiele hierfür sind Minima und Maxima, aber auch die Summen-Funktion:

$$\sum_{n=1}^{100} n = \sum \left(\sum_{n=1}^{10} n, \sum_{n=11}^{20} n, \ldots \sum_{n=90}^{100} n \right)$$

4 https://docs.oracle.com/cd/E37838_01/html/E61035/gpzfw.html#scrolltoc

Da nicht alle Daten zwischengespeichert werden müssen, verbessert der Einsatz von *aggregations* die Skalierung und reduziert den Speicherbedarf erheblich. Vergleichbar mit *hashes* in *perl(1) unterstützt* DTrace nahezu beliebige, auch mehrfache Indizes ohne dabei auf numerische Werte oder Strings beschränkt zu sein. So kann ein Index auch auf Basis des user-stacks gebildet werden:

```
@name[ustack()] = count();
```

Tabelle 1 fasst die seit Solaris 11.3 unverändert verfügbaren *aggregations* zusammen.

Name	**Funktion**
count	zählt die Zahl der Aufrufe
sum	Gesamtsumme der Ausdrücke
min, max	kleinster und größter Wert der Ausdrücke
avg, stddev	arithmetisches Mittel und Standardabweichung
lquantize	lineare Verteilung mit vorgegebenem Intervall und Grenzen
quantize	2^n Verteilung
llquantize	log-lineare Verteilung

Tabelle 1: Aggregations seit Solaris 11.3

Darüber hinaus existieren die in Tabelle 2 zusammengefassten Hilfsfunktionen die der Manipulation und Ausgabe von *aggregations* dienen.

Funktion	**Aufgabe**
trunc(@aggr [, n])	Löscht eine *aggregation* vollständig, bzw. Teile dieser n > 0 erhält die obersten n-Einträge n < 0 erhält die untersten n-Einträge
clear(@aggr)	setzt alle Werte auf Null, Indizes bleiben erhalten
normalize(@aggr, fac)	setzt einen „Normalisierungsfaktor“ jedoch ohne die Werte zu verändern
denormalize(@aggr)	macht die Normalisierung rückgängig

Tabelle 2: Hilfsfunktionen für aggregations

Die Ausgabe von in *aggregations* gespeicherten Daten geschieht mit Hilfe der *printa()* Funktion, die sich stark an *printf()* anlehnt, jedoch automatisch über alle verwendeten Indizes iteriert und vorab eine Sortierung vornimmt. Letztere lässt sich durch die in Tabelle 3 zusammengefassten Optionen steuern.

Option	Aufgabe
aggsortkey	Sortierung der Daten nach Schlüssel bzw. Index. Voreingestellt wird die Ausgabe nach dem Wert der Daten sortiert.
aggsortkeypos	Sofern *aggsortkey* aktiviert wurde kann mit dieser Option die Indexposition spezifiziert werden.
aggsortrev	Kehrt die Reihenfolge um.
aggsortpos	Sofern mehrere aggregations verknüpft innerhalb einer *printa()* Funktion ausgegeben werden, steuert diese Option welche *aggregation* als Basis für die Sortierung herangezogen wird.

Tabelle 3: DTrace Sortieroptionen

Für Beispiele sei auch an dieser Stelle auf die Tagungsbeiträge des Autors aus den vergangenen Jahren verwiesen.

Einige neue DTrace provider in 11.4:
Das Monitoring von System-IO ist sicherlich eine der am häufigsten auftretenden Notwendigkeiten. Hierzu stellte DTrace bereits in der Vergangenheit den *IO-provider* für Plattenzugriffe sowie eine Vielzahl weiterer, etwa Netzwerk spezifischer *provider* bereit. Darüber hinaus hat sich auch der Einsatz des *syscall-providers* für file-IO als nützlich erwiesen.

Der mit 11.4 eingeführte *fileops-provider* erleichtert in allererster Linie das Monitoring von Dateioperationen. Dabei ist es unerheblich ob diese schlussendlich zu echtem Geräte-IO führen oder, etwa im Falle von Lesezugriffen, aus dem Cache bedient werden. Die bereitgestellten *probes* umfassen alle gängigen Dateioperationen, egal ob in der 32- oder 64-Bit Variante:

`chmod, chown, chgrp, close, create, link, mkdir, open, read, readdir, symlink, unlink, write`

Die Daten werden dem *consumer* nur bei erfolgreichem Abschluss der Operation übergeben. Neben dem ersten Argument, einem Zeiger auf eine `fileinfo_t` Struktur, wird als zweiter Wert immer die Latenz der Operation in Form eines `hrtime_t` Types übergeben. *Probe* spezifisch können noch bis zu drei weitere Werte bzw. Zeiger bereitgestellt werden.

```
typedef struct fileinfo {
        string fi_name;         /* name (basename of fi_pathname) */
        string fi_dirname;      /* directory (dirname of fi_pathname) */
        string fi_pathname;     /* full pathname */
        offset_t fi_offset;     /* offset within file */
        string fi_fs;           /* filesystem */
        string fi_mount;        /* mount point of file system */
} fileinfo_t;
```

Ein DTrace-Einzeiler ermittelt beispielsweise die Lese-Latenz bezogen auf den mountpoint.

```
# dtrace -n 'fileops:::read { @[args[0]->fi_mount] = quantize(args[1]); }'
dtrace: description 'fileops:::read ' matched 1 probe
^C

  /zones/cifs/root/smb/data/departement1
           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@@@                     8
           16384 |@@@@@@@@@@@@@@@@@@@@                     8
           32768 |                                         0
           65536 |                                         0
          131072 |                                         0
          262144 |                                         0
          524288 |                                         0
         1048576 |                                         0
         2097152 |                                         0
         4194304 |                                         0
         8388608 |                                         0
        16777216 |@@                                       1
        33554432 |                                         0

  /zones/cifs/root/smb/data/department2
           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@                                        2
           16384 |                                         1
           32768 |                                         1
           65536 |                                         0
          131072 |@@@                                      11
          262144 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     120
          524288 |                                         0
...
```

Leicht verändert bekommen wir die Schreib-Latenz, nun nicht in Nanosekunden sondern sinnvoll in Mikrosekunden, für alle gemounteten ZFS Filesysteme.

```
# dtrace -n 'fileops:::write
    /args[0]->fi_fs == "zfs"/
    { @[args[0]->fi_mount] = quantize(args[1] / 1000); }'
dtrace: description 'fileops:::write ' matched 1 probe
^C
...
  /zones/cifs/root/smb/sw
           value  ------------- Distribution ------------- count
               8 |                                         0
              16 |@@@@@@@@@@@@@@@@@@@@@@@@@                45
              32 |@@@@@@@@@@                               17
              64 |@@@@                                     8
             128 |                                         0
             256 |                                         0
             512 |                                         0
            1024 |                                         0
            2048 |@                                        1
            4096 |                                         0
            8192 |@                                        2
           16384 |                                         0
```

```
  /pool1/home/kiz
           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 12
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 7
            8192 |                                         0

  /pool1/home/student1
           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 34
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 8
            8192 |                                         0
```

Übrigens lassen sich im neuen Solaris Release Filesystem-Latenzen, wenn auch weniger feingranular, wie folgt herausfinden:

```
# fsstat -l zfs 10
 read read   read write write write rddir rddir rddir
  ops bytes  time   ops bytes  time   ops bytes  time
71.4M 1.50T    8n 24.5M  746G  585n  286M 62.4G 1.54u zfs
   98 8.92M    0n   490 6.18M  423n   784 54.9K 4.32u zfs
1.81K 11.8M    4n 1.99K 18.8M  335n   114 46.8K 40.0n zfs
```

Auf Plattenebene hat ein anderes Tool eine entsprechende Erweiterung erfahren:

```
# iostat -Lxn c0t5000C5003C8C82BFd0 10
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    2.4    8.2   18.0   51.8  0.0  0.0    0.0    2.1   0   1
c0t5000C5003C8C82BFd0
latency          range          count      density distribution
                 <16us              0        0.00%        0.00%
               16-32us          74388        0.66%        0.66%
               32-64us         145306        1.29%        1.95%
              64-128us        1833151       16.23%       18.18%
             128-256us        2235358       19.79%       37.97%
             256-512us        2232688       19.77%       57.74%
            512-1024us        2353436       20.84%       78.58%
                 1-2ms         635769        5.63%       84.21%
                 2-4ms         159311        1.41%       85.62%
                 4-8ms         538704        4.77%       90.39%
                8-16ms         733114        6.49%       96.88%
               16-32ms         269782        2.39%       99.27%
               32-64ms          68960        0.61%       99.88%
              64-128ms          12311        0.11%       99.99%
             128-256ms            729        0.01%      100.00%
             256-512ms            106        0.00%      100.00%
            512-1024ms              4        0.00%      100.00%
               >1024ms              0        0.00%      100.00%
                 total       11293117
```

Sicherlich hilfreiche Ergänzungen um einen ersten Eindruck zu erhalten, auch ohne DTrace Kenntnisse.

Sehr hilfreich ist auch eine spannende Erweiterung im Bereich des Netzwerk-Monitoring. Hier klaffte ja bisher eine Lücke zwischen der Datenanalyse wie sie beispielsweise mit *wireshark(8)* erfolgt und der jeweils verantwortlichen Anwendung. Letztere ließ sich mit DTrace bis hin zum IP-Stack nachvollziehen jedoch war die Korrelation der Daten eher mühsame Handarbeit, sofern überhaupt möglich.

Die *pcap(mblk, protocol)* Funktion schließt diese Lücke im Zusammenspiel mit *freopen(filename)*. Im Grunde ähnelt *pcap()* dem Verhalten von *trace()* das ebenfalls Daten in einem Puffer ablegt. Wird die Funktion jedoch im Zusammenspiel mit freopen() verwendet so entstehen eine oder mehrere Dateien die mittels *libpcap* von *wireshark(8)* oder anderen Tools gelesen werden können. Als Voreinstellung sammelt DTrace nur bis zu 2048 Bytes an Daten, d.h. dieser Wert muss bei Bedarf beim Start des Skriptes vergrößert werden.

```
# dtrace -x pcapsize=10000
# dtrace -x DTRACEOPT_PCAPSIZE=10000
```

Alternativ kann dies, sicher die sinnvollere Lösung, als Option im Skript geschehen.

```
#pragma option pcapsize 10000
```

Als bekannte Protokolle sind bereits folgende Konstanten vordefiniert:

```
PCAP_ETHER, PCAP_WIFI, PCAP_PPP, PCAP_IP, PCAP_IPNET, PCAP_IPOIB
```

Damit bleibt, sofern nicht weitere Filter notwendig sind, wie so oft nur eine Zeile übrig. Da *freopen()* eine potentiell destruktive Action ist, muss ihre Verwendung erlaubt werden. Dies geschieht über die Option *-w* auf der Kommandozeile, alternativ als Option im Skript.

```
# dtrace -q -w -n '
    ip:::send {
      freopen("/tmp/cap.%d", pid);
      pcap(args[0]->pkt_addr, PCAP_IP);
      freopen("");
    }'
```

Das Skript legt alle verschickten IP-Pakete pro Prozess-ID in einer eigenen Datei ab.

Falsch ist aber die Annahme, dass eine einfache Erweiterung des Skriptes nun den gesamten Verkehr, also auch den empfangenen, mitschneiden würde:

```
# dtrace -q -w -n '
    ip:::send, ip:::receive {
      freopen("/tmp/cap.%d", pid);
      pcap(args[0]->pkt_addr, PCAP_IP);
      freopen("");
    }'
```

Zu dem Zeitpunkt an dem die *ip:::receive probe* getriggert wird, ist noch nicht bekannt wer der Empfänger des Paketes sein wird. Neben einem Prozess könnte dies auch durchaus der Kernel selber sein. Folgendes kleines Experiment verdeutlicht dies:

```
# dtrace -q -n '
    ip:::send, ip:::receive {
      @[probename, execname, pid] = count();
```

```
    }
    tick-10s {
      printa("%-10s %-12s %5d %@6d\n", @);
      exit(0);
    }'

send       nfsd          1130      1
send       rpcbind        970      1
send       smbd         29769      1
send       sshd         26943      1
send       smbd          8919     12
send       smbd         29861     26
send       smbd          5848     30
send       sshd          7503    162
send       nfsd_kproc    1132   4231
send       sched            0   5995
receive    sched            0  27625
```

Es wird sofort deutlich, dass verschickte IP-Pakete problemlos Prozessen bzw. dem Kernel zuzuordnen sind, auf der anderen Seite alle empfangenen Pakete der Prozess-ID Null und damit erst einmal dem Kernel zugeschlagen werden.

Besser sieht es aus wenn wir von der IP- auf die TCP-Ebene wechseln. Da TCP-Pakete einer Verbindung zugeordnet sind kann DTrace teilweise auf die entsprechenden Daten zurück greifen. Zu beachten ist jedoch, dass lediglich die Prozess-ID, diese wird als *args[1]->cs_pid* bereitgestellt, verlässlich verwendet werden kann.

```
# dtrace -q -n '
    tcp:::send, tcp:::receive {
      @[probename, execname, pid] = count();
    }
    tick-10s {
      printa("%-10s %-12s %5d %@6d\n", @);
      exit(0);
    }'
…
send       smbd         29861     26
receive    sshd          7503     35
send       smbd          5848     37
send       smbd           144    157
send       sshd          7503    171
receive    nfsd_kproc    1132    727
send       nfsd_kproc    1132   1286
receive    sched            0  10984
```

Sofern die Ausgabe, wie im ersten *pcap*-Beispiel, nicht zwingend dynamisch auf individuelle Dateien verteilt werden muss, ist es sehr viel effizienter die Datei zum Startzeitpunkt des Skripts zu öffnen. **Achtung, potentielles Problem:** damit werde alle Ausgaben in diese Datei umgeleitet!

Dies zusammen gefasst startet und überwacht DTrace im folgenden Beispiel das Kommando `nc` und wird damit in die Lage versetzt den kompletten Netzwerkverkehr des Prozesses zu sammeln.

```
trinity # dtrace -q -w -n '
    BEGIN {
      freopen("/tmp/nau.pcap");
```

```
    }

    tcp:::send, tcp:::receive
    /args[1]->cs_pid == $target/ {
      pcap(args[0]->pkt_addr, PCAP_IP);
    }

    END {
      freopen("");
    }' -c "nc -k -l 1234"
```

Schickt man nun von einem anderen System Daten an den „Server“ der auf Port 1234 hört so schneidet DTrace diese mit. Mit *tshark(8)* lassen sich diese dann auswerten.

```
neo# echo "What is the Matrix?" | nc trinity 1234

trinity# tshark -r /tmp/nau.pcap -o data.show_as_text:TRUE -T fields \
             -e frame.number -e ip.src -e ip.dst -e data.text
1     134.60.1.94       134.60.1.113
2     134.60.1.94       134.60.1.113
3     134.60.1.94       134.60.1.113      What is the Matrix?
4     134.60.1.113      134.60.1.94
```

Bei den Ausgabeoptionen von DTrace hat ebenfalls eine hilfreiche Neuerung Einzug gehalten. Mittels *print()* Befehls kann jetzt – in Ergänzung zu *printf()*, *printa()* und *trace()* – in weiten Bereichen der Wert beliebiger Variabler ausgegeben werden ohne dass der jeweilige Typ bekannt sein muss.

Im folgenden soll dies an Hand der *vnode*-Funktion *VOB_LOOKUP* verdeutlicht werden. Sie ist für die Auflösung eines Datei- oder Verzeichnisnamens in eine *vnode*-Struktur verantwortlich und wird im Kernel u.a. vom Systemaufruf *open()* verwendet. In */usr/include/sys/vnode.h* wird sie als

```
extern int fop_lookup(vnode_t *, char *, vnode_t **, struct pathname *,
                      int, vnode_t *, cred_t *, caller_context_t *,
                      int *, struct pathname *
```

deklariert. Dieses Wissen mit DTrace kombiniert:

```
# dtrace -q -n '
    fop_lookup:entry {
      self->vnode = args[0];
    }
    fop_lookup:return
    / self->vnode / {
      print(*self→vnode);
      exit(0);
    }'

vnode_t {
  v_lock = {
    _opaque = [ NULL ]
  }
  v_flag = 0x0
  v_count = 0x2
  v_data = 0xffff8106e5e6c498
  v_vfsp = 0xffffa1c06342e020
```

```
  …
  v_locality = NULL
  v_femhead = NULL
  v_path = "/nau/.cache/mozilla/firefox/1j6xy55g.default/cache2/entries"
```

bzw.

```
# dtrace -q -n '
    fop_lookup:entry {
      self->vnode = args[0];
    }
    fop_lookup:return
    / self->vnode / {
      print(self->vnode->v_path);
      self->vnode = 0;
    }'

char * "/zones/cifs/root"
char * "/zones/cifs/root/smb"
char * "/zones/cifs/root/smb/sw"
char * "/zones/cifs/root/smb/sw/var"
char * "/zones/cifs/root/smb/sw/var/log"
char * "/zones/cifs/root"
^C
```

Analytics und der Solaris StatsStore

Neben den herausragenden Möglichkeiten von DTrace stellt Solaris bereits seit Jahren auch andere Mittel bereit, um das System hinsichtlich der Performance und besonderer Ereignisse (security, …) überwachen zu können. Zu diesen Tools gehören natürlich *kstat* oder der Oldie *vmstat*. Allerdings standen all diese Daten bisher nur in unterschiedlichen Formaten und ohne gemeinsamen Namensraum, also unkorreliert, zur Verfügung. Der mit 11.4 eingeführte StatsStore ändert dies. Er ergänzt die Performance Daten um audit-events und fault-events (FMA), Hardware Konfiguration sowie Informationen über Prozesse und vieles mehr. Gleichzeitig wird ein einheitlicher Namensraum für diese Daten aufgespannt und diese über längere Zeiträume erfasst und ggf. konsolidiert.

Die Steuerung des Systems übernimmt der SMF-Service *svc:/system/sstore* und die Visualisierung wird von *svc:/system/webui/server* erledigt. Auch über die Kommandozeile lassen sich die Daten mit Hilfe von *sstore(1)* in diversen Formaten ausgeben und weiter verarbeiten.

```
# sstore list //:class.disk*
IDENTIFIER
//:class.disk
//:class.disk-controller
```

Die drei im Beispiel ausgewählten Klassen fassen jeweils Ressourcen zusammen die gemeinsame Statistiken ausweisen, hier als Beispiel diejenigen der Klasse `//:class.disk`

```
# sstore list //:class.disk//:stat.*
IDENTIFIER
//:class.disk//:stat.read-bytes
//:class.disk//:stat.read-ops
//:class.disk//:stat.write-bytes
//:class.disk//:stat.write-ops
```

Eine Ressource in diesem Beispiel wären die einzelnen Platten des Systems:

```
# sstore list //:class.disk//:res.*
IDENTIFIER
//:class.disk//:res.name/sd0
//:class.disk//:res.name/sd1
//:class.disk//:res.name/sd10
…

# sstore list //:class.disk//:res.name/sd0//:stat.*
IDENTIFIER
//:class.disk//:res.name/sd0//:stat.read-bytes
//:class.disk//:res.name/sd0//:stat.read-ops
//:class.disk//:res.name/sd0//:stat.write-bytes
//:class.disk//:res.name/sd0//:stat.write-ops
```

Neben Klassen, Ressourcen und Statistiken existiert noch eine vierte Gruppierung, die Partitionen. Sie unterteilt Statistiken, etwa hier am Beispiel der Auslastung von CPU #0:

```
# sstore export -t 2018-09-27T15:07:10 -p 1 \
         //:class.cpu//:res.id/0//:stat.usage//:part.mode
TIME                VALUE IDENTIFIER
2018-09-27T15:07:10  //:class.cpu//:res.id/0//:stat.usage//:part.mode
                    idle: 1199651699.766952
                    intr: 7320396.622533
                    kernel: 115791538.181657
                    user: 13138812.588451
```

Eine detaillierte Beschreibung findet sich in *ssid(7)*.

Die Ausgabe kann auch in anderen Formaten, etwa CSV oder JSON, erfolgen. Auch hier sei lediglich auf die Dokumentation in *sstore.csv(5)* und *sstore.json(5)* verwiesen.

```
# sstore export -t 2018-09-27T15:07:10 -p 2 \
         //:class.cpu//:res.id/0//:stat.usage
TIME                VALUE IDENTIFIER
2018-09-27T15:07:10 1335902447.159593 //:class.cpu//:res.id/0//:stat.usage
2018-09-27T15:07:11 1335903452.350261 //:class.cpu//:res.id/0//:stat.usage

# sstore export -F json -t 2018-09-27T15:07:10 -p 2 \
         //:class.cpu//:res.id/0//:stat.usage
{
     "__version": 1,
     "data": [
          {
               "ssid": "//:class.cpu//:res.id/0//:stat.usage",
               "records": [
                    {
                         "start-time": 1538053630038643,
                         "value": 1335902447.1595931
                    },
                    {
                         "start-time": 1538053631038657,
                         "value": 1335903452.350261
                    }
               ]
          }
```

```
    ]
}
```

sstore(1) ist auch in der Lage Daten zu sammeln, auszugeben und im StatsStore zur späteren weiteren Verwendung abzulegen, hier am Beispiel von IO der Platte *sd0* in 5-Sekunden Intervallen.

```
# sstore capture \
     //:class.disk//:res.name/sd0//:stat.{read-bytes,write-bytes} 5
TIME                VALUE IDENTIFIER
2018-09-28T07:42:48 0      //:class.disk//:res.name/sd0//:stat.write-bytes
2018-09-28T07:42:48 131480 //:class.disk//:res.name/sd0//:stat.read-bytes
2018-09-28T07:42:53 0      //:class.disk//:res.name/sd0//:stat.write-bytes
2018-09-28T07:42:53 131480 //:class.disk//:res.name/sd0//:stat.read-bytes
```

Dauerhaft lassen sich zusammengehörige Gruppen von Statistiken, sogenannte collections, mit Hilfe von `sstoreadm enable-collection …` aktivieren. Eine Übersicht über vordefinierte collections erhält man wie folgt:

```
# sstore info //:class.collection//:collection.*
…
Identifier: //:class.collection//:collection.name/root/cpu-stats
  ssid: //:class.cpu//:stat.context-switches
  ssid: //:class.cpu//:stat.integer-pipe-capacity
  ssid: //:class.cpu//:stat.integer-pipe-usage
```

Das Manual *ssid-collection.json(5)* liefert weitere Details.

Weitaus einfacher gestalten sich die Dinge durch die Nutzung des Web-Interfaces. Dieses präsentiert sich nach der Anmeldung mit einem Dashboard (Abbildung 2). Von hier aus kann dann in weitere Detailansichten navigiert werden, beispielsweise die Memory-Auslastung (Abbildung 3).

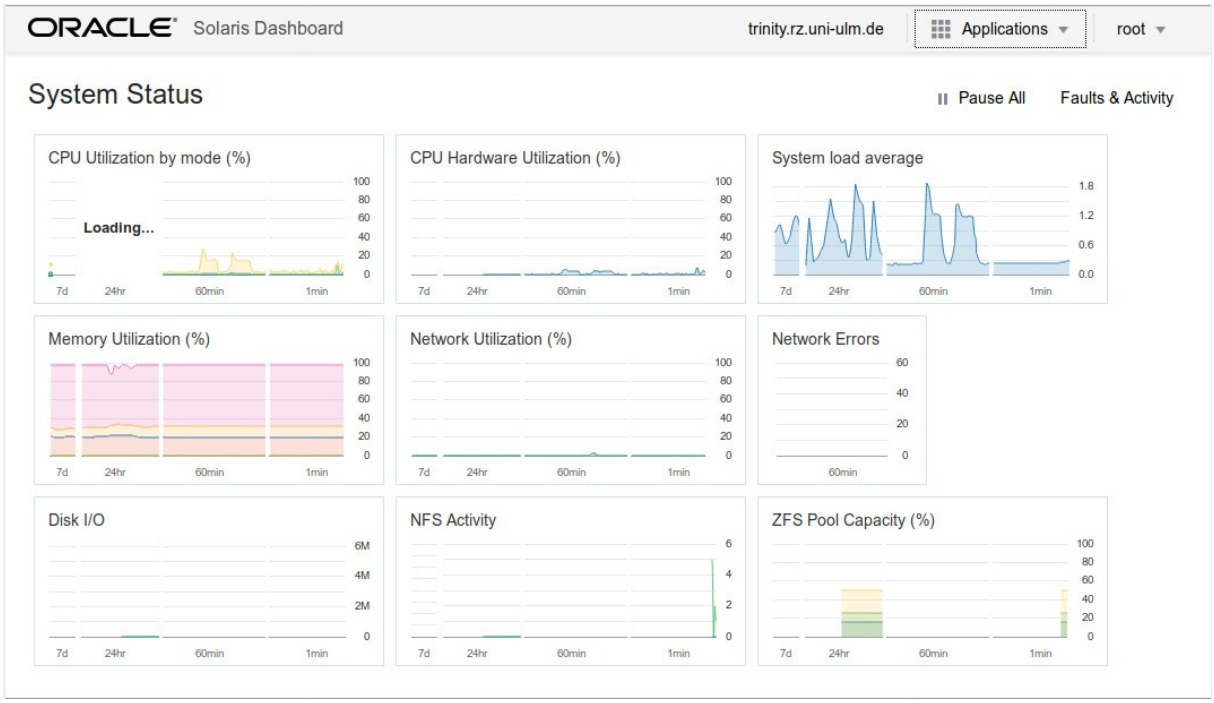

Abbildung 2: Dashboard

In den Kontextmenüs der Graphen können Zeitintervalle ausgewählt werden oder das Sammeln von Daten temporär bzw. auch dauerhaft aktiviert werden. Für eine detaillierte Beschreibung des Web-GUI sowie des StatsStore sei auf die Oracle Dokumentation[5] und auf Blogs[6,7] verwiesen.

5 https://docs.oracle.com/cd/E37838_01/pdf/E56520.pdf
6 https://blogs.oracle.com/solaris/getting-data-out-of-the-statsstore
7 https://blogs.oracle.com/jmcp/solaris-analytics%3a-an-overview

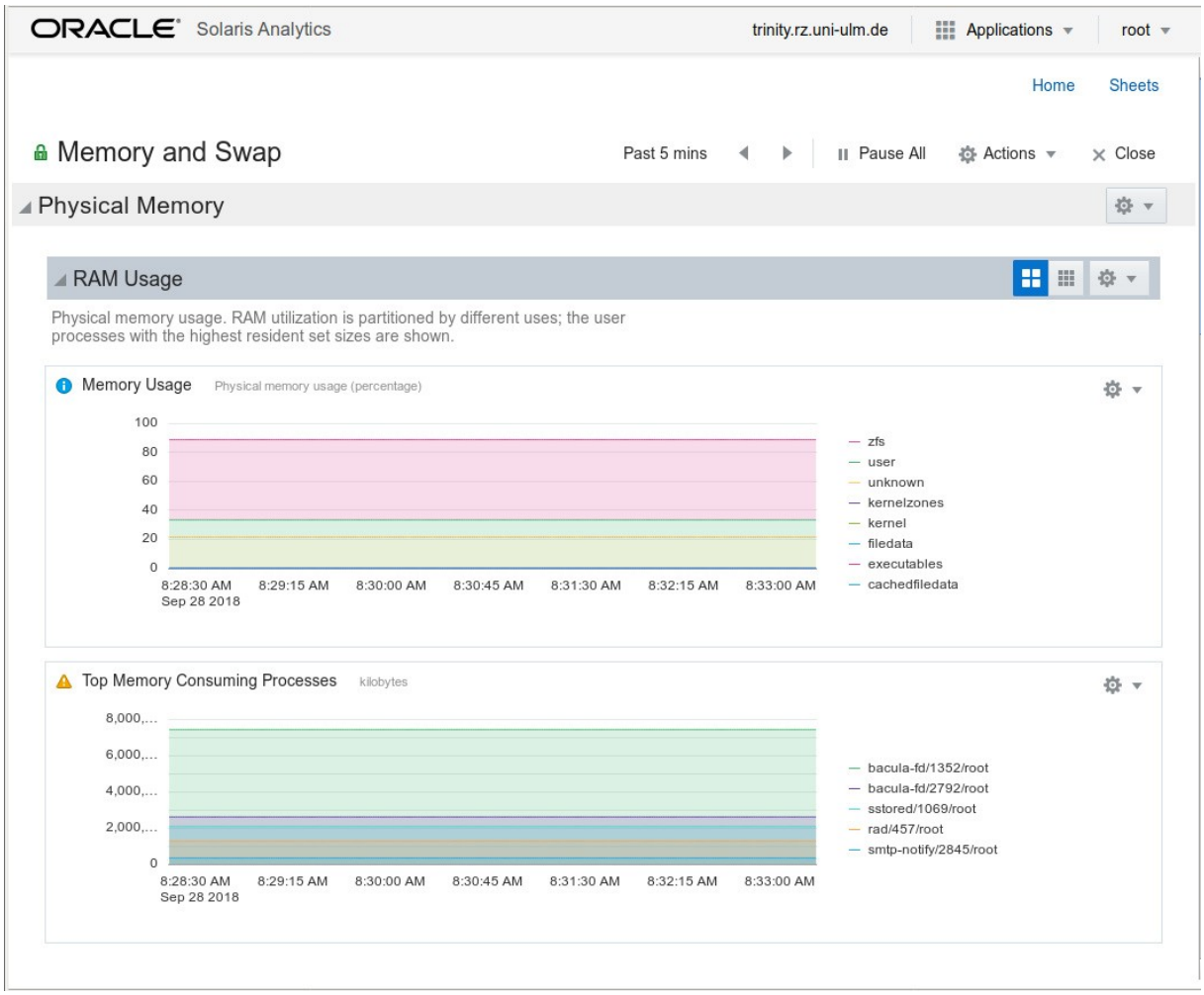

Abbildung 3: Memory Auslastung

Zusammenfassung

Durch die Einführung des StatsStore und die Möglichkeit, hier auch DTrace Skripte einzubinden bietet Solar 11.4 erheblich bessere Monitoring Möglichkeiten als man sie bisher kannte. Dies gilt insbesondere, da sich oft mit wenigen Zeilen D-Code komplexe Analysen erstellen lassen. Es bleibt zu hoffen, dass in Bälde Repositorien oder Blogs hilfreiche Erweiterungen anbieten werden.

Danksagung:

Mein besonderer Dank für die fortlaufende Unterstützung mit Anregungen, Ideen und Korrekturen gilt meinem Kollegen Dr. Harald Däubler.

Kontaktadresse:

Thomas Nau
Universität Ulm – kiz

Albert Einstein Allee 11
D-89081 Ulm

Telefon: +49 (0) 731 50-22464
Fax: +49 (0) 731 50-12-22464
E-Mail Thomas.Nau@uni-ulm.de
Internet: http://www.uni-ulm.de/einrichtungen/kiz