DTrace -
Ein Blick unter die Haube von Solaris

Thomas Nau
Universität Ulm – kiz
Ulm

Schlüsselworte:
DTrace, Performance, Analyse

Einleitung:
Ähnlich wie ZFS hat auch DTrace, die *dynamic tracing facility*, seit dem Erscheinen in Solaris 10 neue Maßstäbe gesetzt – und setzt diese weiterhin mit den Weiterentwicklungen un Solaris 11 – an denen sich auch heute noch Betriebssysteme messen lassen müssen. Fälschlicherweise gehen jedoch viele Anwender aber auch Systemadministratoren davon aus, dass DTrace nur für Kernel-Hacker zu bedienen und zu nutzen ist. Dass dem nicht so ist, lässt sich im folgenden leicht an Hand vieler Beispiele aus dem Bereich der Systemadministration, besonders aber auch aus dem der Entwicklung belegen. So reichen oft wenige Zeilen D, der Programmiersprache von DTrace, um eine Vielfalt an Informationen zu gewinnen die anderweitig oft gar nicht zugänglich wären. Diese beschränken sich hierbei nicht auf CPUs oder das Cache Verhalten, welche oft oben auf der Liste stehen, wenn es um die Analyse von Performance-Engpässen geht, sondern sind viel weitreichender. Wechselwirkungen zwischen mehreren Anwendungen oder auch der Einfluss des Kernels – etwa in Form des IO Sub-Systems – lassen sich einfach messen. Fehlten bis vor wenigen Jahre entsprechend weitreichende Werkzeuge im Repertoire von Solaris, so wurden diese mit der Einführung von DTrace in Solaris 10 bereitgestellt und in Solaris 11 weiter ausgebaut. Speziell im Netzwerkbereich hat DTrace in Solaris 11 wesentliche Erweiterungen erfahren.
Darüber hinaus bietet DTrace auch Entwicklern eine breite Palette von Möglichkeiten, um qualitativ bessere Performancedaten der Anwendung zu ermitteln, die das Gesamtsystem berücksichtigen.

Hintergrund des Autors:
Das Kommunikations- und Informationszentrum (kiz) der Universität Ulm trägt unter anderem die Gesamtverantwortung für deren IT-Infrastruktur, inklusive Telefonie, sowie die Versorgung der Wissenschaftler und Studenten sowohl mit elektronischen als auch mit Print-Medien. Die Kernaufgaben der Abteilung Infrastruktur, deren Leiter der Autor ist, 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-, LDAP-, Portal-, Datenbank- und File-Server der Universität Ulm.

Historie:
Für den Betrieb einer zentralen Infrastruktur sind Hilfsmittel, die im Problem- oder Fehlerfall, aber auch für Planungs- und Monitoring-Zwecke, eine effiziente und effektive Datenerfassung und Auswertung erlauben, unerlässlich. Daher sind entsprechende Tools, jedoch in unterschiedlichster Qualität, seit jeher fester Bestandteil aller UNIX-artigen Systeme bzw. der jeweiligen Distributionen oder Releases. Im Bereich von Solaris zählen hierzu, neben dem *modular debugger mdb(1)*, vor allem die so genannten p- und stat-tools, wie *pstack(1)*, *pfiles(1)*, *prstat(1m)*, *vmstat(1m)* aber auch *truss(1)*. Auch hier lohnt sich ein Blick in die release notes der Solaris updates denn auch hier sind immer wieder nützliche Neuigkeiten zu entdecken. Weniger bekannt ist das sehr mächtige *kstat(1m),* das vollkommen zu unrecht ein Schattendasein fristet, obwohl es den Zugriff auf alle statistischen Daten

des Solaris Kernels ermöglicht. So lassen sich etwa die über das virtuelle Netzwerkinterface (VNIC) scratch0 übertragenen Bytes einfach auslesen.

```
obi-wan# dladm show-vnic scratch0
LINK        OVER      SPEED  MACADDRESS        MACADDRTYPE       VID
scratch0    net0      1000   2:8:20:64:41:d0   random            0

obi-wan# ipadm show-addr scratch0/v4
ADDROBJ            TYPE     STATE        ADDR
scratch0/v4        static   ok           134.60.1.128/24

obi-wan# kstat -p ::scratch0:obytes64 ::scratch0:rbytes64 1 2
link:0:scratch0:obytes64        582189983
link:0:scratch0:rbytes64        11186334449

link:0:scratch0:obytes64        582190251
link:0:scratch0:rbytes64        11186335095
```

Auch Mike Harsch's *arcstat.pl* Tool verwendet die perl-kstat Schnittelle, um Informationen über die ZFS ARC caches zu ermitteln. Zu finden ist es unter
http://blog.harschsystems.com/2010/09/08/arcstat-pl-updated-for-l2arc-statistics/

Im Routinebetrieb sind diesen all Analyse-Tools sind jedoch auch Grenzen gesetzt. So liefert *vmstat(1m)* zwar reichhaltige Informationen über den Speicher oder Kontextwechsel, jedoch ohne diese einzelnen Anwendungen zuzuordnen. Andere Tools wie *prstat(1) bieten e*ine Prozess- bzw. Thread-orientierte Sicht. Diesen fehlt jedoch die Verbindung zum System als Ganzes und die Möglichkeit, die gewonnenen Daten mit denen anderer Analyse-Anwendungen zu korrelieren. Die zugrunde liegende sampling Technik gestattet auch keine sichere Erfassung kurzlebiger Ereignisse.
Abbildung 1 verdeutlicht dies. Das System ist, wie der load-Parameter erkennen lässt, gut ausgelastet, entsprechende Prozesse sind jedoch nicht identifizierbar.

Besonders im IO-Bereich, egal ob Netzwerk oder Storage, sind mit herkömmlichen Solaris Tools nur oberflächliche Informationen zu gewinnen.
Auch das gerne verwendete *truss(1)* hat zwei gravierende Nachteile. Zum einen müssen der Prozess bzw. die Prozesse gestoppt werden um die gewünschten Informationen zu gewinnen. Das Zeitverhalten der Anwendung kann damit drastisch beeinflusst werden und die Erkennung von transienten Fehlern oder Störungen unmöglich machen. Zum anderen ist auch hier eine Korrelation, etwa zur Analyse komplexer Abfolgen wie login-Vorgängen, äußerst aufwändig, bzw. meist sogar unmöglich.

Die Lösung:
DTrace bietet nun auf Grund seiner dynamischen Natur eine Lösung für die meisten der oben genannten Probleme. Es erlaubt eine gleichzeitige Sicht auf die Anwendung als auch auf den Solaris Kernel. Hierzu können sogenannte *probes*, Triggerpunkte, dynamisch sowohl im Kernel, in den zugehörigen Modulen, als auch in Anwendungen und Bibliotheken zur Laufzeit aktiviert oder deaktiviert werden. Dies unterscheidet sich gravierend vom sampling-Ansatz der oben aufgeführten herkömmlichen Tools.
Provider, DTrace Kernkomponenten mit dedizierten Aufgaben, stellen in typischen Solaris 11 Systemen 100.000 und mehr *probes* zur Verfügung. Deren Aktivierung hat im Allgemeinen nur einen zu vernachlässigenden Einfluss auf die Performance des Systems. Etwa 90% dieser *probes* sind dem *function boundary trace provider* (FBT) zuzuordnen und entsprechen den Einsprung- und Rückkehrpunkten der Solaris Kernel Routinen.

```
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
   893 root      211M  204M cpu1     0   10   0:03:02 3.5% catman/1
  1513 nau       101M   35M sleep   59    0   0:00:00 0.3% nautilus/1
  1574 nau        79M   16M sleep   59    0   0:00:00 0.2% gnome-terminal/2
   913 nau        75M   42M sleep   59    0   0:00:01 0.2% Xorg/3
     5 root       0K    0K sleep   99  -20   0:00:03 0.1% zpool-rpool/138
  1511 nau        30M   13M sleep   59    0   0:00:00 0.1% metacity/1
  1512 nau        90M   24M sleep   59    0   0:00:00 0.1% gnome-panel/2
   429 root       13M 3424K sleep   59    0   0:00:01 0.1% nscd/36
  1514 nau        36M   18M sleep   59    0   0:00:00 0.1% isapython2.6/1
  1548 nau        83M   16M sleep   59    0   0:00:00 0.1% mixer_applet2/2
  1519 nau        32M   14M sleep   59    0   0:00:00 0.1% wnck-applet/1
  1546 nau        32M   14M sleep   59    0   0:00:00 0.0% clock-applet/1
  1525 nau        81M   14M sleep   59    0   0:00:00 0.0% gnome-power-man/1
  1536 nau        13M 4644K sleep   59    0   0:00:00 0.0% xscreensaver/1
  1439 nau        21M 7528K sleep   59    0   0:00:00 0.0% gnome-session/2
  1556 root     7416K 5484K sleep   59    0   0:00:00 0.0% hald/4
  1597 nau        11M 3500K cpu0    59    0   0:00:00 0.0% prstat/1
  1522 nau        26M 9040K sleep   59    0   0:00:00 0.0% notification-ar/1
  1507 nau        87M   21M sleep   59    0   0:00:01 0.0% gnome-settings-/1
    13 root       23M   22M sleep   59    0   0:00:15 0.0% svc.configd/23
  1500 nau        14M 5664K sleep   59    0   0:00:00 0.0% gconfd-2/1
   274 root     3588K 1936K sleep   59    0   0:00:00 0.0% dbus-daemon/1
Total: 93 processes, 435 lwps, load averages: 1.22, 0.66, 0.28
```

Abbildung 1: prstat Beispiel

Die Abbildung 2 verdeutlicht die Integration in den Solaris Kernel.

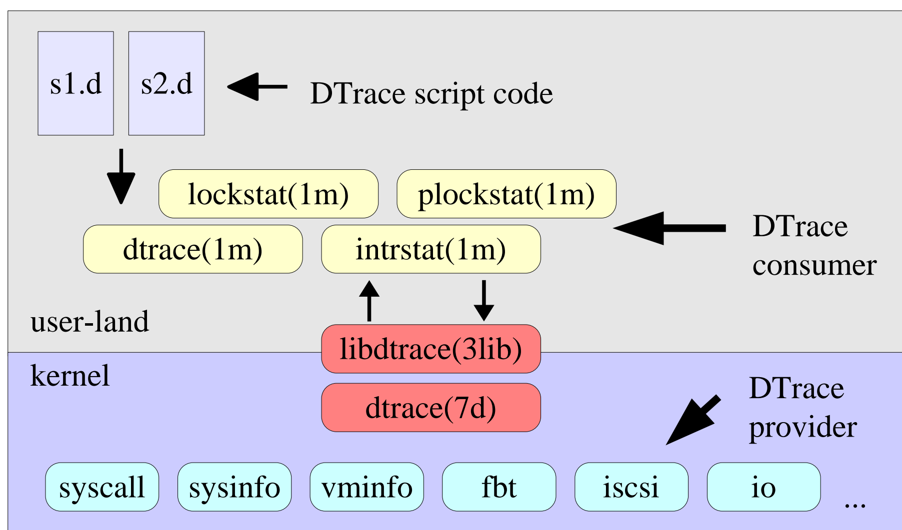

Abbildung 2: DTrace Kernel Schnittstelle

Datenfluss:
Innerhalb des Kernels übernehmen die *provider* die sinnvolle Aufarbeitung der gesammelten Daten und stellen sie in Puffern bereit. Neben den üblichen Datenstrukturen lassen sich auch Speicherinhalte sowie Kernel- und User-Stacks sichern. Die Größe und Organisation der Datenbereiche innerhalb des Kernels, etwa in Form von Ring-Puffern, sind individuell konfigurierbar. Es ist zu beachten, dass

Daten ggf. zwischen den unterschiedlichen Adressräumen der Anwendungen und des Kerns kopiert werden müssen. Die notwendigen Funktionen stellt DTrace zur Verfügung. Ein Schreibzugriff auf Daten ist aus Sicherheitsgründen nur sehr eingeschränkt möglich, da dies die Datenintegrität sowie die Sicherheitsfunktionen des Kernels gefährden kann.

Neue *provider* werden im Rahmen von Solaris Major-Releases zur Verfügung gestellt. So ist der *ip-provider* Bestandteil von Solaris 11 und nicht in Version 10 verfügbar. Als die im Allgemeinen am häufigsten genutzten sind zu nennen:

io
Die *probes* stellen detaillierte Informationen über IO-Operationen auf Platten, Bändern aber auch NFS-Servern zur Verfügung. Darunter sind neben File- und Gerätenamen auch Größe und Richtung der Datenübertragung zu finden.

ip
Anwendungen, wie etwa *snoop(1m), tcpdump(1)* oder auch *wireshark(1m)* erlauben zwar eine weitgehende Protokoll-Analyse, jedoch nicht immer eine eindeutige Zuordnung zu Prozessen. Diese bietet der *ip-provider* und stellt aufbereitete Daten des IP-headers, etwa IP-Adressen in Form eines Strings, zur Verfügung. Damit stellt er ein mächtiges Werkzeug zur Server- oder Client-seitigen Analyse der Netzlast dar.

syscall
Stellt *probes* für Einsprung- und Rückkehrpunkte von Systemaufrufen bereit.

proc
Alle Informationen bezüglich der Erzeugung von Prozessen und Threads, sowie die Abarbeitung von Signalen werden von den *probes* des *proc-providers* erfasst und bereitgestellt. Eine Analyse der oben genannten login-Vorgänge ist hiermit einfach möglich.

sched
Die *probes* des *sched-providers* dienen oft als Ergänzung zu den Daten des *proc-providers,* da sie Informationen über das Laufzeitverhalten aus Sicht des Kerns enthalten. Dazu gehören das CPU scheduling-Verhalten, aber auch Thread-Synchronisationsmechanismen, wie sie im Bereich von OpenMP Anwendungen zu finden sind.

profile
Dieser *provider* ist keiner im herkömmlichen Sinn, sondern gestattet DTrace-Anwendungen die periodische Ausführung von Befehlen. Damit lassen sich sowohl sampling als auch die regelmäßige Ausgabe von Daten à la *vmstat(1m)* realisieren.

Neben dem *ip-provider* stellt Solaris 11 speziell im Netzwerk Umfeld weitere *provider* zur Verfügung und trägt damit den Anforderungen Rechnung, die im Bereich der Storage-Virtualisierung entstehen. Zu diesen zählen die Protokoll spezifischen *ip-, tcp-, udp-, iscsi-, nfsv3-, nfsv4-* und der *srp-provider*. Auch hier kommt wieder eine besondere Stärke des DTrace Konzeptes zum Tragen: der Anwender muss sich nicht um die Aufbereitung der Daten kümmern sondern bekommt zum Beispiel IP-Adressen direkt als String zur Weiterverarbeitung geliefert. Darüber hinau bietet DTrace in aktuellen Solaris Versionen auch den Zugriff auf die seit langem in Systemen vorhandenen CPU hardware performance counter. Hierdurch wird die Analyse von Korrelationen zwischen Anwendung, also Software, und Hardware Ereignissen wie z.B. der Anzahl von cache misses spürbar vereinfacht. Auf Grund der

Hardware nahen Natur dieses *providers* existieren einige Randbedingungen hinsichtlich Verfügbarkeit von *probes* und Zahle der gleichzeitigen *consumer*. Details finden sich in der Solaris Dokumentation: http://docs.oracle.com/cd/E26502_01/html/E28556/cpc-provider.html

Detaillierte Beschreibungen der aktuellen in Solaris 11.1 verfügbaren *provider* finden sich unter http://docs.oracle.com/cd/E26502_01/html/E28556/gkyal.html

Die *consumer*, so auch das *dtrace(1m)* Tool, sind für das Lesen der Daten aus den vom Solaris Kern bereitgestellten Puffern verantwortlich und nutzen hierzu einen Gerätetreiber und die Routinen der Bibliothek *libdtrace*. Der Zugriff auf die Daten geschieht hierbei asynchron. Neben *dtrace(1m),* dem sicherlich bekanntesten *consumer,* setzen immer mehr andere Solaris Werkzeuge, wie in Abbildung 2 ersichtlich, auf DTrace auf.

Die Skript-Sprache "D":
Die DTrace-Technologie bedient sich einer simplen, an "C" angelehnten Skriptsprache namens "D", um Anwendern einen einfachen und vertrauten Zugang zu bieten. Das *dtrace(1m) Kommando* steuert das Compilieren von Skripten, lädt diese in den Kernel und agiert gleichzeitig als *consumer* für die anfallenden Daten.
Als Datentypen stehen alle in C gängigen Typen zur Verfügung, wenngleich gelegentlich unter anderen Bezeichnungen. Der Typ *uintptr_t* ist ein Beispiel hierfür. Neben üblichen arithmetischen und logischen Operationen glänzt D insbesondere durch die integrierte String-Verarbeitung, assoziative Arrays – vergleichbar mit *perl* Hashes – und vor allem Aggregationen, einem der mächtigsten DTrace Features. Dazu später mehr. Eine große Menge vordefinierter Variablen bietet einfachen Zugriff etwa auf Prozess-IDs, credentials, Zeiten und weiteres mehr. Hervorzuheben ist das *fds[]* array das die einem file-descriptor zugehörigen Daten bereitstellt. Damit lässt sich auch bei *read()* oder *write()* Operationen die zugehörige Datei o.ä. ermitteln.
Da der Code im Kernel ausgeführt wird, stellt D weder Sprungbefehle noch Schleifen-Konstrukte zur Verfügung um die Stabilität des Systems nicht zu gefährden.

D-Skripte bestehen aus Blöcken, die sequentiell von oben nach unten abgearbeitet, besser ausgedrückt, überprüft werden. Da ein D-Skript kein Hauptprogramm im eigentlichen Sinn enthält, agieren die einzelnen Code-Blöcke eher im Sinne von Unterprogrammen, die von den entsprechenden DTrace Routinen im Kernel verwendet, also aufgerufen werden. Jeder dieser Blöcke setzt sich aus einer *probe* Definition, einer optionalen Bedingung sowie den auszuführenden Aktionen zusammen. Die Nachbildung eines *if-then-else* Konstruktes verdeutlicht dies:

```
/* if Zweig */
probe1, probe2, ...
/ Bedingung /
{
    Aktionen
}

/* else Zweig */
probe1, probe2, ...
/ ! (Bedingung) /
{
    Aktionen
}
```

Die Namens-Syntax zur Definition von *probes* folgt dem Schema

```
provider:module:function:name
```

also etwa

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

"*", "?" und "[…]" können in der gewohnten Weise als Platzhalter verwendet werden:

```
syscall::open*:
syscall:*:open*:*
syscall::open: , syscall::open64:
```

`dtrace -l` liefert eine Liste aller verfügbaren *probes,* die bei Bedarf mittels `-n Suchmuster` eingeschränkt werden kann.

```
obi-wan# dtrace -l -n \*close\*
   ID   PROVIDER      MODULE          FUNCTION NAME
13957        sdt      sv              sv_lyr_close sv_lyr_close_recursive
13990        sdt      sv              sv_lyr_close sv_lyr_close_end
15178     fsinfo      genunix         fop_close close
16942      nfsv4      nfssrv          rfs4_op_close op-close-done
17021        sdt      nfssrv          rfs4_do_state_hydrate nfss-i-osclosed
17099      nfsv4      nfssrv          rfs4_op_close op-close-start
17293        sdt      sppptun         sppptun_close sppptun-client-close
```

Ein erstes Skript:

Das Anzeigen aller Systemaufrufe für eine spezifizierte Anwendung lässt sich sehr leicht mit dem folgenden einfachen Skript erreichen:

```
obi-wan# cat syscalls.d

#!/usr/sbin/dtrace -s
#pragma D option quiet

syscall:::entry
/ execname == $$1 /
{
    printf("%-20s %6d %s\n", execname, pid, probefunc);
}

obi-wan# ./syscalls.d imapd
imapd                  1490 gtime
imapd                  1490 pollsys
imapd                 10875 gtime
imapd                 10875 gtime
…
```

execname, *pid* und *probefunc* sind vordefinierte DTrace-Variablen, die zur Laufzeit mit den entsprechenden Daten initialisiert werden. Die letztgenannte *probefunc* enthält den Namen des Systemaufrufs. Da bei der Definition von den wildcard Mechanismen Gebrauch gemacht wurde stellt dies eine einfache Möglichkeit dar die unterschiedlichen Systemaufrufe zu unterschieden. `$1` wird durch das erste Kommandozeilen Argument, hier `imapd`, ersetzt, wobei eine Maskierung des Dollar-Zeichens bei diesem Einsatzzweck zwingend ist. Zu guter Letzt unterdrückt `#pragma D option quiet` zusätzliche Statusausgaben und erleichtert hierdurch das Parsen der Ausgabe bzw. verbessert deren Lesbarkeit.

Auch die Zahl der empfangenen IP Pakete pro IP Adresse lässt sich leicht ermitteln. Gleichzeitig demonstriert das Beispiel eine weitere Stärke von DTrace. Durch die effiziente Gestaltung lassen sich die meisten Aufgaben mit Einzeilern, also direkt auf der Kommandozeile, erledigen:

```
obi-wan# dtrace -n 'tcp:::receive { @[args[2]->ip_saddr] = count(); }'
dtrace: description 'tcp:::receive ' matched 4 probes
^C

  134.60.1.15                    1
  134.60.1.42                    3
  134.60.60.100                 24
```

Aggregationen:
Aggregationen nutzen eine Besonderheit mancher mathematischer Funktionen aus. Dabei liefert die Anwendung der Funktion auf eine Teilmenge der Daten im ersten Schritt und nachfolgend auf die Zwischenergebnisse im zweiten Schritt das selbe Ergebnis wie die Anwendung der Funktion auf die Gesamtmenge der Daten. Die Summen-Funktion, Minima und Maxima sind Beispiele für derartige mathematische Funktionen. Der Einsatz von Aggregationen hält den Speicherbedarf gering, da keine Notwendigkeit besteht, alle Daten zu speichern. Außerdem werden Probleme mit der Skalierung hinsichtlich der potentiellen Menge der Daten vermieden.
Der in DTrace verwendbare Index für Aggregationen ist nahezu beliebig und kann neben numerischen Werten oder Zeichenketten (`execname`, `pid`, …) insbesondere auch aus dem Aufrufstack der Anwendung, `ustack()`, oder des Kernels, `stack()`, bestehen.

```
@name[ keys ] = aggfunc ( args );
```

Derzeit sind in Solaris 11.1 die folgenden Aggregationen verfügbar:

- count — zählt die Zahl der Aufrufe
- sum — Gesamtsumme der Ausdrücke
- avg — arithmetischer Mittelwert
- min, max — kleinster/größter Wert der Ausdrücke
- stddev — Standardabweichung
- lquantize — lineare Verteilung mit vorgegebenem Intervall und Grenzen
- quantize — 2^n Verteilung

Insbesondere die letzten beiden Verteilungsfunktionen erweisen sich bei der nachfolgend gezeigten Analyse von IO Problemen als hilfreich.

Tipp:
Die Sortierung von Aggregationen bei der Ausgabe – eine leider wenig dokumentierte Möglichkeit – lässt sich mit DTrace Optionen steuern, etwa

```
setopt("aggsortkey", true);
```

Folgende Optionen sind derzeit hierfür definiert:

- aggsortkey — Sortierung nach Index, nicht nach Wert
- aggsortrev — Umkehrung der Reihenfolge
- aggsortkeypos — Nummer des Index, nach dem sortiert werden soll

Für den Fall, dass gleichzeitig mehrere Aggregationen ausgegeben werden sollen, kann diejenige, die als Referenz für die Sortierung fungieren soll, ebenfalls ausgewählt werden:

- aggsortpos Nummer der Aggregation, die als primärer Schlüssel dient

IO Analyse:
Mit herkömmlichen, wie am Anfang des Artikels beschriebenen Tools, ist eine live Analyse des IO Verhaltens von Anwendungen nicht realisierbar. Häufige unbeantwortete Fragestellungen sind:

1. Welche Dateien werden bearbeitet?
2. Welche Größenverteilung weisen meine IO-Operationen bzw. Netzwerkpakete auf?
3. Wie lange dauern Schreib- oder Leseoperationen?
4. Welcher Prozess ist für die mit *iostat(1m)* sichtbare hohe Bandbreite verantwortlich?

DTrace hat auf diese und viele andere Fragen schnelle und eindeutige Antworten. Die zugehörigen Skripte umfassen in aller Regel weniger als 20 Zeilen.

Frage1: Welche Anwendung schreibt/liest welche Dateien?

Die Antwort auf diese Frage kann über mehrere DTrace Mechanismen beantwortet werden. Deren Wahl hängt von den gewünschten Informationen ab. Ist lediglich die logische Sicht, als diejenige auf Dateien, interessant so reicht meist der Ansatz über die zugehörigen Systemaufrufe. So liefert das Skript:

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

BEGIN
{
    printf("%-10s %5s %8s %8s %s\n",
        "application", "pid", "function", "bytes", "file");
}

syscall::write*:entry,
syscall::read*:entry
/ fds[arg0].fi_fs == "zfs" /
{
    printf("%-10s %5d %8s %8d %s\n",
        execname, pid, probefunc, arg2, fds[arg0].fi_pathname);
}
```

folgende Informationen:

```
application   pid function  bytes file
init         2382     read   1024 /data/host1/zone/root/etc/inittab
init         2382     read   1024 /data/host1/zone/root/etc/inittab
init         2382     read   1024 /data/host1/zone/root/etc/inittab
init         2382     read   1024 /data/host1/zone/root/etc/inittab
sshd        26375     read   1024 /data/host1/zone/root/etc/gss/mech
sshd        26375     read   1536 /data/host1/zone/root/etc/krb5/krb5.conf
^C
```

Wie bei DTrace üblich enthält *execname* den Namen des Prozesses. Sind jedoch auch die zu Grunde liegenden Platten o.ä. von Belang so bietet sich die Nutzung des io-*providers* an. Dieser stellt in seiner

start-probe Strukturen bereit (vgl. C-struct), aus denen sich sowohl die Namen der Dateien und Geräte als auch die Art der Operation, Schreiben bzw. Lesen, sowie die Anzahl der Bytes auslesen lassen.

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

io:::start {
    @iostats[args[1]->dev_statname, args[2]->fi_name, execname] =
       sum(args[0]->b_bcount);
}

END {
    printf("%10s %20s %15s %10s\n", "DEVICE", "FILE", "APP", "BYTES");
    printa("%10s %20s %15s %10@d\n", @iostats);
}
```

Der Name der Anwendung wird gemeinsam mit einigen anderen Parametern als Index für die Aggregation namens `iostats` verwendet. Die Funktion `printa()` übernimmt einfach und elegant die Ausgabe der Daten bei Beendigung des Skriptes. Die interne DTrace *probe* `END` findet hierzu Verwendung. Im Gegensatz zum ersten Beispiel erhalten wir hier also aggregierte Daten.

```
obi-wan# ./file_io.d
^C
DEVICE                 FILE             APP      BYTES
 nfs12              489.dat           sched      32768
 nfs12               478.dx           sched      32768
  ssd0   cis_Pd_complex.chk        l502.exe    5980160
  ssd2   cis_Pd_complex.chk        l502.exe    6356992
  ssd2          pt_slab.DM   siesta-constr    7946240
  ssd0          pt_slab.DM   siesta-constr    7995392
  md10   cis_Pd_complex.chk        l502.exe   12337152
  md10          pt_slab.DM   siesta-constr   15941632
  ssd0        Gau-3413.rwf        l502.exe   16588800
  ssd2        Gau-3413.rwf        l502.exe   17686528
  md10        Gau-3413.rwf        l502.exe   34275328
...
```

Frage 2: Welche Größenverteilung weisen meine Netzwerkpakete auf?

Diese Frage lässt sich mit DTrace nur in Solaris 11 beantworten da der notwendige *ip-provider* erst ab dieser Version verfügbarist. Das nachfolgende Skript basiert auf einem Beispiel des DTrace Wiki http://wikis.oracle.com/display/DTrace/ip+Provider

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

ip:::send {
    @ipstats[args[2]->ip_daddr, execname] = quantize(args[2]->ip_plength);
}
```

Wie schon der *io-provider* stellt auch der *ip-provider* in seiner *send-probe* Zeiger auf Strukturen mit aufbereiteten Daten zur Verfügung. So zeigt etwa `args[2]` auf die `ipinfo_t` Struktur:

```
typedef struct ipinfo {
        uint8_t ip_ver;         /* IP version (4, 6)   */
        uint16_t ip_plength;    /* payload length      */
        string ip_saddr;        /* source address      */
        string ip_daddr;        /* destination address */
} ipinfo_t;
```

Siehe auch http://docs.oracle.com/cd/E26502_01/html/E28556/glhgu.html

Gut zu erkennen ist die Aufbereitung der Daten durch DTrace etwa die Bereitstellung von IP Adressen als Strings. Die Ausgabe von Aggregationen erledigt *dtrace(1m)* automatisch wenn das Skriptes beendet wird. Wie im obigen Beispiel ersichtlich ist hierzu keine explizite Anweisung notwendig.

```
192.168.23.23                                         nfsd
    value  ------------- Distribution ------------- count
       16 |                                         0
       32 |                                         1
       64 |                                         0
      128 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              244
      256 |@@@@@@@@@@                               88
      512 |                                         0
     1024 |@@@@                                     32
     2048 |                                         0
...
```

Deutlich sichtbar ist die relativ geringe Größe einer Vielzahl von NFS bezogenen Paketen.

Eine Erweiterung des Konzeptes unter Verwendung des *profile-providers* liefert mit wenig Aufwand periodische top-ten Ausgaben der aktivsten Kommunikationspartner in Echtzeit.

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

BEGIN {
    ts = timestamp;
}

ip:::send {
    @ipstats[args[2]->ip_daddr, probename] = sum(args[2]->ip_plength);
}

ip:::receive {
    @ipstats[args[2]->ip_saddr, probename] = sum(args[2]->ip_plength);
}

/* print top-n normalized to bytes per second; reset stats when done */
tick-$1 {
    trunc(@ipstats, $2);
    normalize(@ipstats,(timestamp-ts) / 1000000000);
    printf("\n%Y\n", walltimestamp);
    printa("%-15s %-10s %@15d\n", @ipstats);
    ts = timestamp;
    trunc(@ipstats);
}
```

Die beiden *probes send* und *receive* finden in Erweiterung des ersten Beispiels Verwendung. Mit ihnen ist es einfach möglich, die Richtung der Datenpakete zu bestimmen. Diese wird in Form der Variable `probename` in der Aggregation als Schlüssel hinterlegt. Darüber hinaus wertet das Skript zwei ihm übergebene Parameter aus. Diese werden als `$1` und `$2` referenziert. Der erste Parameter definiert die Periode, der zweite die Anzahl der auszugebenden Werte. Dazu wird die Aggregation mit dem Befehl `trunc(@ipstats, $2)` auf die gewünschte Länge gekürzt. Etwas schwieriger gestaltet sich die Normierung auf "Bytes pro Sekunde", da der für die *tick-probe* notwendige Parameter eine Einheit, etwa `sec`, enthalten muss. Damit scheidet dieser Parameter für die Verwendung in arithmetischen Operationen aus. Das Problem ist leicht durch einen eigene Zeitmessung zu lösen. Hierzu wird beim Start des Skripts die globale Variable `ts` mit Hilfe einer weiteren internen *probe*, `BEGIN`, initialisiert und dann für weitere Berechnungen verwendet.
`timestamp` repräsentiert einen Zähler mit Nanosekunden Auflösung. Da dieser im Gegensatz zu `walltimestamp` jedoch keinen definierten Startzeitpunkt besitzt ist `timestamp` lediglich für die Messung von Zeitdifferenzen verwendbar. Die Formatierungsoption `%Y` wandelt Zeitangaben in leicht lesbare Strings um.

```
obi-wan# ./ip_top.d 2sec 5

2012 Sep 12 14:29:36
134.60.70.171   send                    64
134.60.1.111    send                   284
134.60.1.111    receive                642
134.60.60.100   receive               3232
134.60.60.100   send                  3424

2012 Sep 12 14:29:38
134.60.1.3      receive                 24
134.60.1.15     receive                 30
134.60.1.15     send                   274
134.60.60.100   receive               2222
134.60.60.100   send                  2354
^C
```

Auch die anderen Fragen lassen sich vergleichbar einfach und elegant mit Hilfe von wenigen Zeilen DTrace-Code beantworten.

Es mag nun der Eindruck entstanden sein, DTrace wäre in erster Linie ein Werkzeug für SysAdmins, doch dies ist mitnichten so. Auch ohne Zugriff auf den Quellcode lassen sich Laufzeit-Analysen von Bibliotheken oder Anwendungen einfach und elegant durchführen. Auch ein Debugger ist hierzu nicht notwendig.
Der dazu notwendige *pid-provider* nimmt eine Sonderstellung ein, da seine *probes* erst zur Laufzeit dynamisch erzeugt werden und an einen Prozess geknüpft sind. Dabei kann es sich um einen bereits laufenden Prozess handeln, oder einen, der unter der Kontrolle von DTrace neu erzeugt wird. Vorsicht ist bei der genauen Spezifikation der *probe* geboten, da der *pid-provider* Befehle bis auf Assembler-Ebene verfolgen kann. Beispiele für eine solche *probe* Definition für einen Prozess mit der pid 54321 sind:

```
pid54321:my-object:my-function:8
pid54321:libc.so.1:strcpy:entry
pid54321:libc.so:strcpy:entry
pid54321:libc:strcpy:entry
```

Mit dem folgenden Skript lässt sich beispielsweise die Zeit messen, die eine Anwendung in einzelnen Funktionen, sowohl denjenigen der Anwendung als auch von Funktionen in externen Bibliotheken, zubringt. Um eine möglichst flexible Nutzung des Skriptes zu gewährleisten, werden auch hier die Namen der Bibliothek sowie der gewünschten Funktionen als Kommandozeilen Parameter übergeben. Wildcards, '*' und '?', können verwendet werden, sind aber entsprechend zu maskieren, damit sie nicht bereits von der ausführenden shell interpretiert werden.

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

pid$target:$1:$2:entry {
    /* "self" variables use thread-local storage as we might
     * look at a process with more than a single thread
     *
     * vtimestamp holds the time the thread was actually
     * running on a CPU without wait times
     */
    self->ts = vtimestamp;
}

/* check if self-ts has been initialized to prevent
 * from race conditions
 */
pid$target:$1:$2:return
/ self->ts / {
    @stats[probemod, probefunc] = sum(vtimestamp -self->ts);
    self->ts = 0;
}

END {
    printa("%10@dns %12s:%s\n", @stats);
}
```

Für ein mit parallelisiertes Beispielprogramm ergibt sich bezüglich der OpenMP spezifischen Routinen folgendes Bild:

```
obi-wan# ./lib_timing.d -c "./partest 10 10" libmtsk ""
...
     63204ns libmtsk.so.1:spin_unlock
     69472ns libmtsk.so.1:spin_lock
     91216ns libmtsk.so.1:libmtsk_info_init
    105080ns libmtsk.so.1:barrier_init
    158324ns libmtsk.so.1:memmanage_init
    160816ns libmtsk.so.1:threads_fini
    184148ns libmtsk.so.1:memmanage_fini
    358240ns libmtsk.so.1:sleep_at_barrier
    922020ns libmtsk.so.1:slave_wait_for_work
```

Auch Aufrufhierarchien sind mit dieser Technik einfach darstellbar. DTrace unterstützt dies explizit durch die Kommandozeilen Option '-F'. Anhand der Verwendung von Funktionen der C-Bibliothek (*libc.so)* durch das Kommando *date* soll dies nachfolgend veranschaulicht werden.

```
#!/usr/sbin/dtrace -s

pid$target:libc.so::entry,
pid$target:libc.so::return
{
    /* use "automatic printing" feature */
}
```

```
obi-wan# ./libc_trace.d  -F -c date

dtrace: script './libc_trace.d' matched 5789 Probes
Mon Aug 15 16:31:17 MET 2011
dtrace: pid 16648 has exited
CPU FUNCTION
  5    -> lmalloc
  5      -> getbucketnum
  5      <- getbucketnum
  5      -> initial_allocation
  5        -> __systemcall
  5        <- __systemcall
  5      <- initial_allocation
  5    <- lmalloc
...
```

Der Erweiterung der DTrace Funktionalitäten zur Analyse von Anwendungen sind nahezu keine Grenzen gesetzt sofern der Quellcode zur Verfügung steht. Die notwendigen Tools, um eigene *provider* in die Anwendung zu integrieren, gehören zur Grundausstattung von Solaris. Verdeutlichen lässt sich dies durch das folgende kleine C-Programm das in einer Schleife einzelne Zeichen vom Eingabegerät liest.

```
obi-wan# cat keypress.c

#include <stdio.h>
#include <sys/sdt.h>

int main (int argc, char *argv[]) {
    int c;

    while ((c = getchar()) != EOF) {
        DTRACE_PROBE1(keypress, read, c);

        /* my application code starts here */
    }
}
```

Um sie eigene Anwendung "DTrace fähig" zu machen, sind lediglich die im Beispiel rot markierten Ergänzungen notwendig:

- Einbinden der Header Datei `<sys/sdt.h>`
- Verwendung der vordefinierten passenden Makros, abhängig von der Anzahl der Parameter, um die gewünschten *probes* zu erzeugen und Daten zu übergeben

Darüber hinaus benötigt DTrace noch einige Informationen bzgl. der neuen, selber definierten, *provider*. Diese finden sich unter http://docs.oracle.com/cd/E26502_01/html/E28556/gkydr.html

```
obi-wan# cat keypress_d.d

Provider keypress {
    Probe read(int);
};

#pragma D attributes Evolving/Evolving/Common Provider keypress Provider
#pragma D attributes Private/Private/Common   Provider keypress module
#pragma D attributes Private/Private/Common   Provider keypress function
#pragma D attributes Evolving/Evolving/Common Provider keypress name
#pragma D attributes Evolving/Evolving/Common Provider keypress args
```

Mit diesen Informationen lässt sich nun eine ausführbare Anwendung erzeugen und anschließend unter Kontrolle von DTrace starten.

```
obi-wan# cc -O -c keypress.c

obi-wan# dtrace -32 -G -s keypress_d.d keypress.o

obi-wan# cc -O -o keypress keypress_d.o keypress.o -ldtrace

obi-wan# cat keypress_d.d keypress.c |
  dtrace -q -c ./keypress -n \
    'keypress$target:::read { @ = lquantize(arg0,0,255,32);}'
```

Als Ergebnis erhält man eine Übersicht über die Verteilung der ASCII Codes in den Eingabedateien `keypress_d.d` und `keypress.c` in Gruppen à 32 Bytes. Schön zu erkennen ist die Häufung von Kleinbuchstaben ab dem ASCII Code 96.

```
     value  ------------- Distribution ------------- count
       < 0 |                                         0
         0 |@@                                       40
        32 |@@@@@@@@@                                174
        64 |@@@                                      51
        96 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@             550
       128 |                                         0
```

Eine weitere Stärke spielt DTrace in Kombination mit anderen Tools aus die die Ausgabe von DTrace weiter verarbeiten. Visualisierungstools wie beispielsweise *graphviz* eignen sich hervorragend um Zusammenhänge einfach darzustellen. Hierbei erzeugt das DTrace Skript die *dot* Eingabedaten. Mit wenigen Zeilen D-Code lassen sich beispielsweise komplexe Prozesshierarchien (Abbildung 3) schnell und übersichtlich darstellen.

In Brendan Gregg's Blog finden sich eine Vielzahl weitergehender Informationen etwa zum Visualisierung mit Hilfe von heatmaps.
http://dtrace.org/blogs/brendan/2012/03/26/subsecond-offset-heat-maps/

```
#!/usr/sbin/dtrace -s

proc:::exec {
    self->parent = execname;
}

proc:::exec-success
/ self->parent != NULL / {
    @c[self->parent, execname] = count();
    self->parent = NULL;
}

END {
    printf("digraph ExecPaths{\n");
    printa("  \"%s\" -> \"%s\" [weight=%@d];\n",@c);
    printf("}\n");
}
```

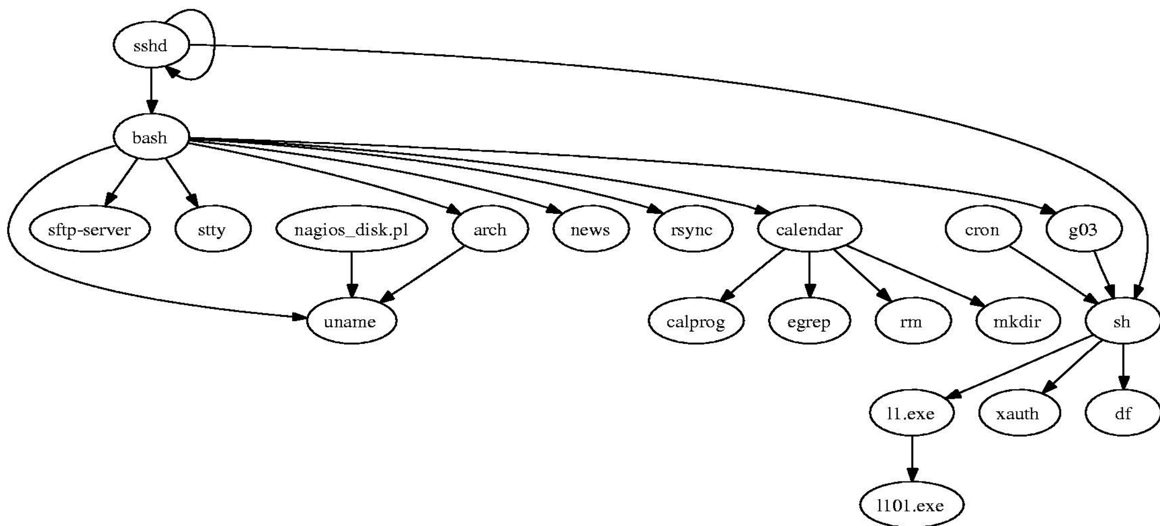

Abbildung 3: Prozesshierarchien

Es bleibt zu erwähnen, dass alle vorgestellten Techniken ohne root-Rechte nutzbar sind. Im Rahmen des mit Solaris 10 eingeführten Rechte-Managements (least privileges) lassen sich einzelnen Anwendern, etwa auch Entwicklungsgruppen, die notwendigen Privilegien zuweisen. Damit steht der Nutzung dieses mächtigen Tools, und dessen Integration in eigene Anwendungen, nichts mehr im Weg.

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