DTrace für Alle -
Performance-Analyse für Entwickler und SysAdmins

Thomas Nau
Universität Ulm – kiz
Ulm

Schlüsselworte:

DTrace, Performance

Einleitung:

CPUs, caches oder compiler-Optionen sind oft der erste Gedanke, wenn es um die Analyse von Performance-Engpässen geht. Die Wechselwirkung zwischen mehreren Anwendungen wird hier häufig ebenso vernachlässigt wie der Einfluss des Kernels, etwa in Form des IO Sub-Systems. Fehlten bis vor wenigen Jahre entsprechend weitreichende Werkzeuge im Repertoire von Solaris, so wurden diese mit der Einführung von DTrace, der *dynamic tracing facility*, in Solaris 10 bereitgestellt.
Entgegen der landläufigen Annahme, DTrace wäre nur etwas für SysAdmins, bietet es auch Entwicklern eine breite Palette von Möglichkeiten, sofern diese etwa neben Performancedaten der Anwendung auch eine Sicht auf das Gesamtsystem benötigen, um das Verhalten der eigenen Applikation besser analysieren zu können.

Hintergrund des Autors:

Das Kommunikations- und Informationszentrum (kiz) der Universität Ulm trägt unter anderem die Gesamtverantwortung für deren IT-Infrastruktur, inklusive Telefonie, sowie deren Versorgung 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 auf HA-Cluster basierenden Mail-, LDAP-, Portal-, Datenbank- und File-Server der Universität.

Historie:

Der Betrieb derartiger Server erfordert Hilfsmittel, die eine schnelle und effiziente Problemanalyse erlauben. Daher sind in vielen UNIX-artigen Systemen entsprechende Tools seit vielen Jahren fester Bestandteil der Distributionen. 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)*. Wenig bekannt, fristet häufig insbesondere das sehr mächtige *kstat(1m)* ein Schattendasein, obwohl es den Zugriff auf alle statistischen Daten des Solaris Kernels ermöglicht. Beispielsweise die über das Netzwerkinterface e1000g0 übertragenen Bytes.

```
obi-wan# kstat -p ::e1000g0:obytes64 ::e1000g0:rbytes64 1 3
e1000g:0:e1000g0:obytes64       13236488622698
e1000g:0:e1000g0:rbytes64       3709464336358

e1000g:0:e1000g0:obytes64       13236489042229
e1000g:0:e1000g0:rbytes64       3709464651554
```

All diesen Analyse-Tools sind jedoch im Routinebetrieb auch Grenzen gesetzt. So liefert *vmstat(1m)* zwar reichhaltige Informationen über den Speicher oder Kontextwechsel, jedoch ohne diese einzelnen Anwendungen zuzuordnen. Eine Prozess- bzw. Thread-orientierte Sicht bieten wiederum andere Tools, etwa *prstat(1)*. Diesen fehlt dafür im Gegenzug die Sicht auf das System als Ganzes und auch die Möglichkeit, die Daten mit denen anderer Analyse-Anwendungen zu korrelieren. Auch erlaubt die zu Grunde liegende sampling Technik keine sichere Erfassung kurzlebiger Ereignisse.
Der nachfolgende *prstat(1)* screenshot verdeutlicht dies. Das System ist, wie der load-Parameter erkennen läßt, gut ausgelastet, entsprechende Prozesse sind jedoch nicht identifizierbar.

```
jedi
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
   937 nau      8644K 2568K sleep   10    0   0:00:25  13% loop.sh/1
   445 root       11M 3652K sleep   59    0   0:00:04 2.4% nscd/44
   698 nau        67M   35M sleep   59    0   0:00:07 0.3% Xorg/3
   912 nau        76M   18M sleep   59    0   0:00:00 0.3% gnome-terminal/2
   870 nau        27M   16M sleep   59    0   0:00:00 0.1% metacity/1
  6919 nau      8988K 3148K cpu0    59    0   0:00:00 0.0% prstat/1
   873 nau        12M 4976K sleep   59    0   0:00:00 0.0% xscreensaver/1
   872 nau        90M   30M sleep   49    0   0:00:00 0.0% nautilus/1
   884 nau        28M   16M sleep   59    0   0:00:00 0.0% wnck-applet/1
   865 nau        29M   17M sleep   59    0   0:00:00 0.0% gnome-settings-/1
   871 nau        80M   21M sleep   59    0   0:00:00 0.0% gnome-panel/1
 11614 nau      7432K 1148K sleep   59    0   0:00:00 0.0% sleep/1
   897 nau        78M   18M sleep   59    0   0:00:00 0.0% mixer_applet2/2
   510 root       10M 1756K sleep   59    0   0:00:00 0.0% VBoxService/7
     5 root        0K    0K sleep   99  -20   0:00:00 0.0% zpool-rpool/136
  3833 nau      8680K 2808K sleep   59    0   0:00:00 0.0% bash/1
   520 root     4608K 3304K sleep   59    0   0:00:00 0.0% console-kit-dae/2
   693 root     9944K 3352K sleep   59    0   0:00:00 0.0% gdm-binary/2
   197 root     2132K 1480K sleep   59    0   0:00:00 0.0% pfexecd/3
   248 root     2548K 1432K sleep   60  -20   0:00:00 0.0% zonestatd/5
   345 root     7288K  952K sleep   59    0   0:00:00 0.0% iscsid/2
   163 daemon   7352K 1244K sleep   59    0   0:00:00 0.0% kcfd/2
   552 root     8152K 1468K sleep   59    0   0:00:00 0.0% automountd/2
Total: 85 processes, 420 lwps, load averages: 1.07, 0.61, 0.29
```

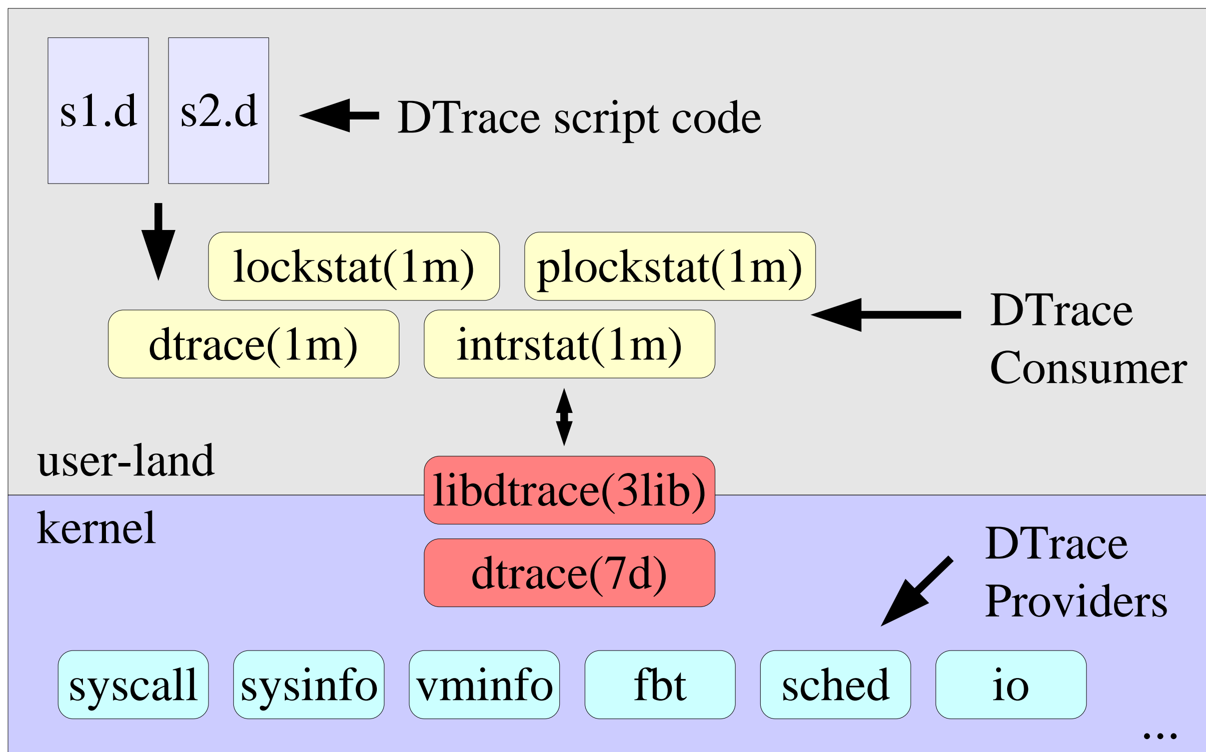
Im IO-Bereich, egal ob Netzwerk oder Storage, sind ausschließlich oberflächliche Informationen mit Bordmitteln zu erhalten.
Auch das gerne verwendete *truss(1)* hat zwei gravierende Nachteile: Zum einen müssen der oder die Prozesse gestoppt werden, um die Daten zu gewinnen, was das Zeitverhalten ändert und damit die Erkennung von transienten Problemen oft unmöglich macht. Zum anderen ist auch hier eine Korrelation, etwa zur Analyse komplexer Abfolgen wie login-Vorgängen, äußerst aufwändig, bzw. zuweilen sogar unmöglich.

Neue Welt:

DTrace bietet nun auf Grund seiner dynamischen Natur eine Lösung für die meisten der oben genannten Probleme und erlaubt insbesondere auch eine Sicht sowohl auf die Anwendung als auch gleichzeitig auf den Solaris Kernel. Hierzu können sogenannte *probes*, Triggerpunkte, dynamisch sowohl im Kernel, den zugehörigen Modulen, als auch in Anwendungen und Bibliotheken zur

Laufzeit aktiviert oder deaktiviert werden. Dies unterscheidet sich gravierend vom sampling-Ansatz der oben genannten herkömmlichen Tools.
Provider, DTrace Kernkomponenten mit dedizierten Aufgaben, stellen in typischen Systemen 75.000 und mehr *probes* zur Verfügung. Deren Aktivierung hat im Allgemeinen nur einen zu vernachlässigenden Einfluss auf die Performance des Systems. Noch zu erwähnen bleibt, dass ca. 90% der *probes* dem *function boundary trace provider* (FBT) zuzuordnen sind, der die Einsprung- und Rückkehrpunkte der Unterroutinen des Solaris Kerns instrumentiert.

Die nachfolgende Abbildung verdeutlicht die Integration in den Solaris Kernel.



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 jedoch zu beachten, dass Daten ggf. zwischen den unterschiedlichen Adressräumen der Anwendungen und des Kerns kopiert werden müssen. Die notwendigen Funktionen stellt DTrace jedoch zur Verfügung. Ein Schreibzugriff auf Daten ist aus Sicherheitsgründen nur sehr eingeschränkt möglich.

Neue *provider* werden im Rahmen von neuen Solaris Releases zur Verfügung gestellt. So ist der nachfolgend genannte *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 ala *vmstat(1m)* realisieren.

Weiter gehende Beschreibungen, auch für neue bzw. experimentelle provider finden sich unter
http://wikis.sun.com/display/DTrace/Providers

Die *consumer*, so auch das *dtrace(1m)* Tool, sind für das Lesen der Daten verantwortlich und nutzen hierzu einen Gerätetreiber und die Routinen der Bibliothek *libdtrace*. Der Zugriff geschieht asynchron. Ist *dtrace(1m)* sicherlich der bekannteste *consumer,* so setzen doch immer mehr andere Solaris Werkzeuge, wie in der Grafik ersichtlich, auf DTrace auf.

D:

Um Anwendern einen einfachen Zugang zu offerieren, bedient sich die DTrace-Technologie einer simplen, an "C" angelehnten Skriptsprache namens "D", sowie des Kontrolltools *dtrace(1m)*. Dieses steuert das Übersetzen von Skripten und agiert gleichzeitig als *consumer* für die anfallenden Daten.
Als Datentypen stehen alle in C gängigen Typen zur Verfügung, wenngleich gelegentlich unter anderen Bezeichnungen, etwa *uintptr_t*. Neben üblichen arithmetischen und logischen Operationen glänzt D insbesondere durch die integrierte String-Verarbeitung, assoziative Arrays (vgl. *perl* Hashes) und Aggregationen. Dazu später mehr. Eine große Menge vordefinierter Variablen bietet einfachen Zugriff etwa auf Prozess-IDs, credentials, Zeiten und vieles mehr.
Um die Stabilität des Systems nicht zu gefährden, denn der Code wird im Kernel ausgeführt, stehen jedoch weder Schleifen-Konstrukte noch Sprungbefehle zur Verfügung.

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 *providern* 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.

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-Variable, die zur Laufzeit mit den entsprechenden Daten initialisiert werden. Die letzte der drei, *probefunc*, beinhaltet den Namen des Systemaufrufs, da bei der Definition von den wildcard Mechanismen Gebrauch gemacht wurde. `$1` wird durch das erste Kommandozeilen Argument, hier `imapd`, ersetzt, wobei eine Maskierung des Dollar Zeichens zwingend ist. Zu guter Letzt unterdrückt `#pragma D option quiet` zusätzliche Statusausgaben.

Aggregationen:

Aggregationen nutzen eine mathematische Besonderheit mancher Funktionen aus, bei der die Anwendung auf Teilmengen der Daten und nachfolgend auf die Zwischenergebnisse dasselbe Ergebnis liefert, wie die Anwendung der Funktion auf die Gesamtmenge der Daten. Die Summenfunktion sowie Minima und Maxima sind Beispiele hierfür. Der Einsatz derartiger Funktionen hält den Speicherbedarf gering, da keine Notwendigkeit besteht alle Daten zu speichern. Außerdem werden Probleme mit der Skalierung vermieden. Der in DTrace verwendbare Index für Aggregationen ist nahezu beliebig also etwa auch die Aufrufstacks `ustack()` und `stack()` oder `execname`, `pid`, ...

Derzeit sind die folgenden Aggregationen verfügbar:

- count   zählt die Zahl der Aufrufe
- sum   Gesamtsumme der Ausdrücke
- avg   arithmetischer Mittelwert
- min, max   kleinster/größter Wert der Ausdrücke
- lquantize   lineare Verteilung
- quantize   2^n Verteilung

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

IO Analyse:

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

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 vergleichbare Fragen eindeutige und schnelle Antworten, wobei die zu Grunde liegenden Skripte in aller Regel weniger als 20 Zeilen umfassen.

Frage1: Welche Anwendung schreibt/liest welche Dateien?

```
#!/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 *io-provider* von DTrace 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. Wie bei DTrace üblich enthält *execname* den Name des Prozesses. Dieser 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.

```
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 derzeit leider nur in OpenSolaris oder Solaris 11 Express mit DTrace beantworten, da der notwendige ip-provider nur dort verfügbar ist. Das nachfolgende Skript basiert auf einem Beispiel unter http://wikis.sun.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);
}
```

Analog zum *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 sogenannte `ipinfo_t` Struktur (entnommen aus http://wikis.sun.com/display/DTrace/ip+Provider):

```
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;
```

Gut sichtbar ist die Aufbereitung der Daten, etwa die Umwandlung von IP Adressen, die bereits als Strings vorliegen. Eine automatische Ausgabe aller Aggregationen geschieht durch dtrace(1m) automatisch bei Beendigung des Skriptes. Eine explizite Anweisung ist hierzu nicht notwendig, wie obiges Beispiel demonstriert..

```
...
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 sehr schnell, und in Echtzeit, periodische top-ten Ausgaben der aktivsten Kommunikationspartner.

```
#!/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);
}
```

In Erweiterung des ersten Beispiels finden hier zwei *probes*, *send* und *receive* Verwendung. Damit ist es einfach möglich, die Richtung der Datenpakete zu bestimmen. In Form der Variable `probename` wird dies auch in der Aggregation als Schlüssel hinterlegt. Des weiteren wertet das Skript zwei ihm übergebene Parameter aus, die als `$1` und `$2` referenziert werden. Der erste definiert die Periode, der zweite die Anzahl der auszugebenden Werte. Hierzu wird mittels `trunc(@ipstats, $2)` die Aggregation lediglich auf die entsprechende Länge gekürzt. Schwieriger gestaltet sich die Normierung auf Bytes pro Sekunde, da der für die tick-probe notwendige Paremeter zwingend eine Einheit, etwa also `10sec`, enthalten muss und damit für die Verwendung in arithmetischen Operationen ausscheidet. Leicht lösbar ist dieses Problem, indem die Zeit gestoppt wird. Hierzu wird beim Start des Skripts die globale Variable `ts` mit Hilfe einer weiteren internen *probe*, `BEGIN`, initialisiert und für weitere Berechnungen verwendet. Die Funktion `timestamp` repräsentiert einen Zähler mit Nanosekunden Auflösung, jedoch ohne definierten Startzeitpunkt. Daher ist er im Gegensatz zu `walltimestamp` lediglich für Differenzmessung verwendbar. Erwähnenswert ist noch die Formatierungsoption `%Y,` die Zeitangaben in leicht lesbare Strings umwandelt.

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

2009 Jun  4 16:35:36
134.60.84.144   receive             34276
134.60.84.170   receive             49792
134.60.215.64   receive             54448
134.60.1.50     receive            301724
134.60.40.100   receive           1304200

2009 Jun  4 16:35:38
134.60.84.131   send               101752
134.60.84.170   receive            124928
134.60.84.170   send               303596
134.60.1.50     receive            408576
134.60.2.117    receive            580712
^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. So lassen sich ebenso einfach Laufzeit-Analysen von Bibliotheken oder Prozessen durchführen, ohne dass der Zugriff auf den Quellcode notwendig ist oder ein Debugger verwendet wird.

Der im nachfolgenden Skript zum Einsatz kommende *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 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 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
```

```
#!/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);
}
```

Damit ergibt sich beispielsweise für ein OpenMP Programm:

```
obi-wan# OMP_NUM_THREADS=2 \
             ./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 einfach darstellbar, wie etwa die Verwendung von Funktionen der C-Bibliothek (*libc.so)* durch das Kommando *date* zeigt:

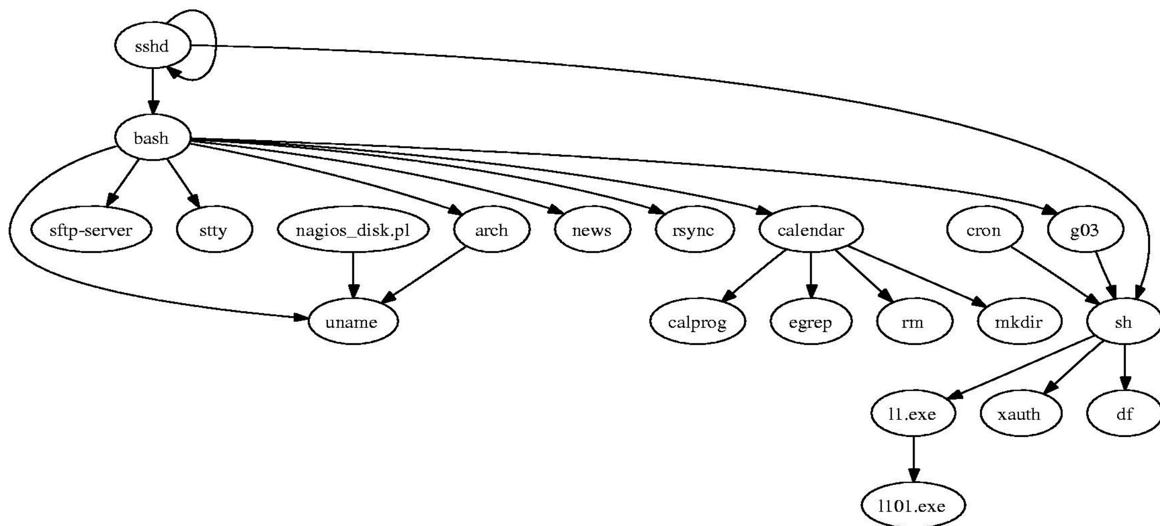
```
#!/usr/sbin/dtrace -s

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
...
```

Eine weitere, besondere Stärke spielt DTrace in Kombination mit Visualisierungstools wie *graphviz* aus, wo wenige Zeilen Code etwa zur Darstellung von Prozesshierarchien ausreichen:



Bleibt zu erwähnen, dass alle vorgestellten Techniken ohne root-Rechte nutzbar sind. Im Rahmen des mit Solaris 10 eingeführten Rechte-Managements lassen sich z.B. Entwicklern 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 50-22471
E-Mail Thomas.Nau@uni-ulm.de
Internet: http://www.uni-ulm.de/einrichtungen/kiz