DTrace -
Die Antwort auf (fast) alle Fragen

Thomas Nau
Universität Ulm – kiz
Ulm

Schlüsselworte:
DTrace, Performance, Analyse

Einleitung:
DTrace, die *dynamic tracing facility*, hat seit ihrem Erscheinen in Solaris 10 vor annähernd 10 Jahren neue Maßstäbe im Bereich der System Analyse und des Performance Monitoring gesetzt. Auch heute noch ist DTrace das Tool an dem sich sowohl Betriebssysteme, als auch deren Werkzeuge und Distributionen messen lassen müssen. Jedoch gehen viele Systemadministratoren, Anwender und Entwickler immer noch fälschlicherweise 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 Performance Analyse und 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. Im Zeitalter der Virtualisierung und von Big Data stehen hierbei nicht mehr CPUs oder das Cache-Verhalten im Vordergrund, wenn es um die Analyse von Performance-Engpässen geht. Vielmehr sind es die weitreichenden Wechselwirkungen zwischen den virtualisierten Gastsystemen, ihren Anwendungen und besonders auch der Einfluss des Kernels vor allem in Form des IO-Sub-Systems und des Netzwerk-Stacks. Fehlten bis vor wenigen Jahren entsprechend weitreichende Werkzeuge sowohl im Repertoire von Solaris, als auch von anderen UNIX Varianten, so wurden diese mit der Einführung von DTrace in Solaris 10 bereitgestellt, seither weiter ausgebaut und insbesondere an die Weiterentwicklungen im Solaris 11 Kernel – etwa dem überarbeiteten Netzwerk-Stack – angepasst.
Darüber hinaus bietet DTrace auch Entwicklern eine breite Palette von Möglichkeiten, um qualitativ bessere Performancedaten der Anwendung zu ermitteln, da diese das Gesamtsystem berücksichtigen. Deutlich wird dies im „Backup“ Showcase auf Seite 16.

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-, LDAP-, Portal-, Datenbank- und File-Server der Universität Ulm.
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 Landeshochschulnetzes BelWü[2].

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

Historie:
Hilfsmittel, die eine effiziente und effektive Datenerfassung und Auswertung für das Performance-Monitoring zu Planungszwecken aber auch zur Analyse im Problem- oder Fehlerfall ermöglichen, sind für den Betrieb einer zentralen IT-Infrastruktur unerlässlich. Daher sind entsprechende Tools, wenngleich in unterschiedlichster Qualität, seit jeher fester Bestandteil aller für Server zum Einsatz kommenden Betriebssysteme bzw. der jeweiligen Distributionen oder Releases. Im Bereich von Solaris zählen hierzu, neben dem *modular debugger mdb(1)*, vor allem die lange verfügbaren sogenannten p- und stat-tools, wie *pstack(1)*, *pfiles(1)*, *prstat(1m)*, *vmstat(1m)* aber auch *truss(1)*.
Auf alle Fälle lohnt sich ein Blick in die release notes der Solaris Releases und Updates (SRU), denn auch hier sind immer wieder nützliche Neuigkeiten und Erweiterungen zu entdecken. Dazu gehören etwa *ctstat(1)* und *fsstat(1m)* oder im Netzbereich *dlstat(1m), flowstat(1m)* und *ipstat(1m*).
Meist unbekannt ist das sehr mächtige *kstat(1m),* das vollkommen zu unrecht, ein Schattendasein fristet. Es ermöglicht den Zugriff auf alle statistischen Daten des Solaris Kernels über eine C-API, perl-Module oder mittels Kommandozeile. Das Beispiel zeigt eine einfache Möglichkeit, die über ein virtuelles Netzwerkinterface (VNIC), das etwa einer bestimmten Zone oder Service zugeordnet ist, übertragenen Bytes zu ermitteln. In unserem Fall gehört die VNIC zum IMAP-replication-service. Es ist zu beachten, dass es sich um absolute Werte handelt.

```
obi-wan# dladm show-vnic
LINK         OVER         SPEED  MACADDRESS        MACADDRTYPE VIDS
imaprepl0    net0          1000   2:8:20:ea:f6:b4   random      0

obi-wan# ipadm show-addr imaprepl0
ADDROBJ           TYPE     STATE        ADDR
imaprepl0/v4      static   ok           134.60.1.28/24

obi-wan# kstat -p ::imaprepl0:obytes64 ::imaprepl0:rbytes64 1 2
link:0:imaprepl0:obytes64       254797953722
link:0:imaprepl0:rbytes64       391165973353

link:0:imaprepl0:obytes64       254797955526
link:0:imaprepl0:rbytes64       391165984296
```

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/

```
jedi# arcstat.pl 5
    Time  read  miss  miss%  dmis  dm%  pmis  pm%  mmis  mm%  arcsz     c
13:52:06   50M  909K      1  542K    1  366K   43  601K    1   150G  254G
13:52:11    1K     0      0     0    0     0    0     0    0   150G  254G
...

jedi# kstat -p ::arcstats:data_size ::arcstats:c 5
zfs:0:arcstats:data_size        152601377600
zfs:0:arcstats:c                273776410624

zfs:0:arcstats:data_size        152611827584
zfs:0:arcstats:c                273776410624
```

Im Routinebetrieb sind diesen herkömmlichen Analyse-Tools jedoch auch Grenzen gesetzt. So liefert *vmstat(1m)* zwar reichhaltige Informationen über den Speicher oder Kontextwechsel, jedoch ohne

diese einzelnen Anwendungen zuzuordnen. Dagegen bieten andere Tools wie zum Beispiel *prstat(1)* eine reine Prozess- bzw. Thread-orientierte Sicht. Damit fehlt ihnen die Verbindung zum System als Ganzes und dadurch insbesondere auch die Möglichkeit, die gewonnenen Daten mit denen anderer Analyse-Anwendungen zu korrelieren. Die zu Grunde liegende sampling Technik bringt eine weitere Einschränkung mit sich, da sie keine sichere Erfassung kurzlebiger Ereignisse gestattet.
Abbildung 1 verdeutlicht dies. Das System ist, wie der load-Parameter erkennen lässt, gut ausgelastet, entsprechende einzelne Prozesse oder UIDs sind jedoch nicht identifizierbar.

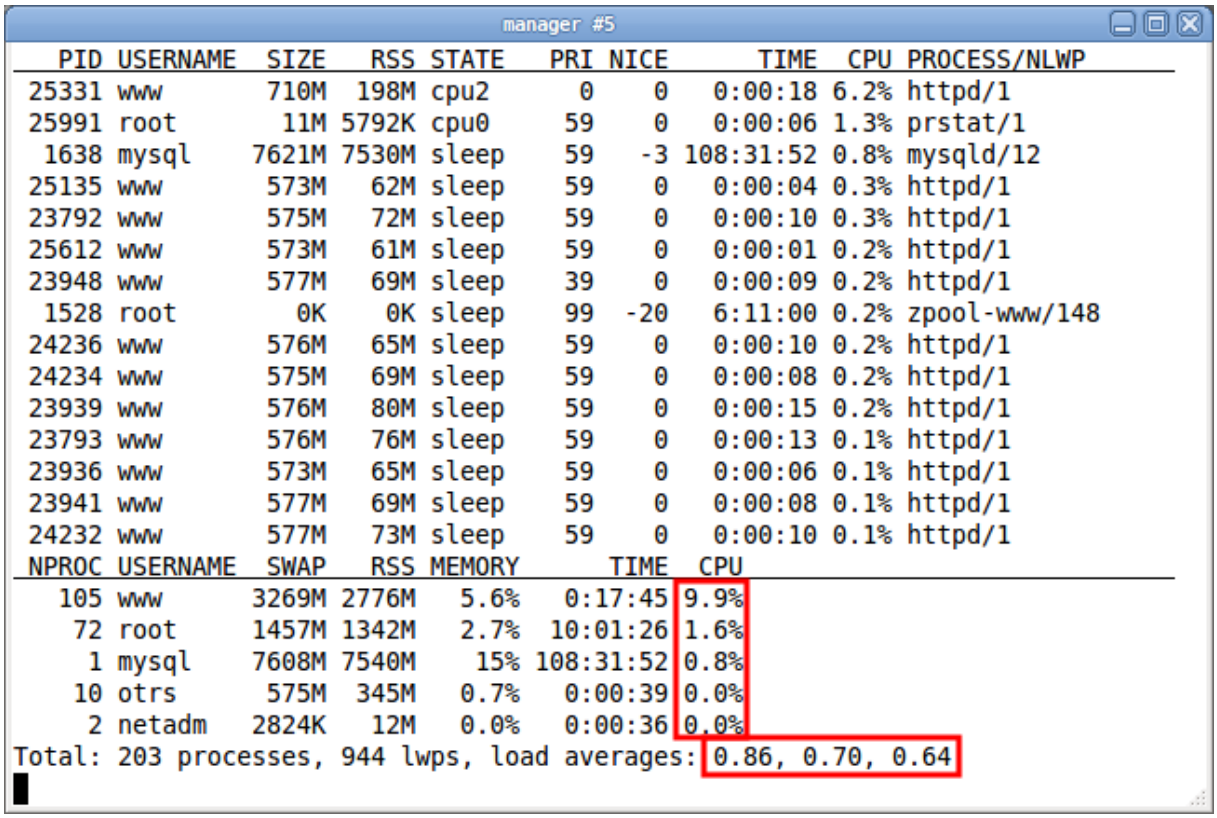

Abbildung 1: prstat Beispiel

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 gravierende Nachteile. Prozesse müssen 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. *truss(1)* bietet keine Möglichkeit, die Ergebnisse einzelner Messungen zu korrelieren um komplexe Abfolgen wie etwa login-Vorgänge zu analysieren. Dies kann allenfalls im Nachgang mit Hilfe eigener Skripte erreicht werden.

Die Lösung:

> *„A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.“*[3]

3 „Erste Regel von Mentat“ aus „Dune – Der Wüstenplanet“

DTrace bietet nun auf Grund seiner dynamischen Natur eine Lösung für die meisten der oben genannten Probleme. Es erlaubt eine gleichzeitige Sicht sowohl auf die Anwendung als auch auf den zugrunde liegenden 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 üblicher Tools.
Provider – DTrace Kernkomponenten mit dedizierten Aufgaben – stellen bei Solaris 11 Systemen ca. 100.000 oder 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 der die Einsprung- und Rückkehrpunkte der Solaris Kernel Routinen wiederspiegelt

Abbildung 2 verdeutlicht die Integration in den Solaris Kernel.

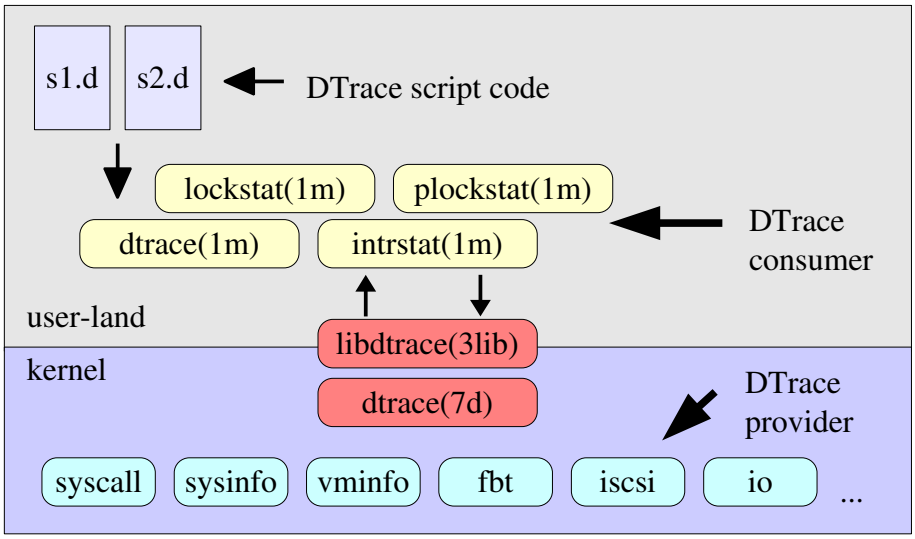

Abbildung 2: DTrace user-land / Kernel Schnittstelle

Besonders erwähnenswert ist, dass alle vorgestellten Techniken auch ohne root-Rechte nutzbar sind. Das mit Solaris 10 eingeführte Rechte-Management „least privileges" erlaubt es, einzelnen Nutzern und Gruppen die notwendigen Privilegien zuzuweisen.

Datenfluss:
Innerhalb des Kernels übernehmen die *provider* die Aufgabe, die gesammelten Daten sinnvoll aufzubereiten und in Puffern für die weitere Verwendung bereit zu stellen. So werden etwa IP-Adressen als String und nicht in Binärform bereitgestellt. 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 sind die *cpc-, ip-, tcp-, udp-* und *iscsi-provider* Bestandteil von Solaris 11 und nicht in Version 10 verfügbar. Auch hier lohnt ein Blick in die release notes. Im Allgemeinen am häufigsten genutzten werden:

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, wie erwähnt, 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 protokollspezifischen *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 hinaus 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 hardwarenahen Natur dieses *providers* existieren einige Randbedingungen hinsichtlich Verfügbarkeit von *probes* und Anzahl der gleichzeitig unterstützten *consumer*. Details finden sich unter:
http://docs.oracle.com/cd/E36784_01/html/E36846/cpc-provider.html

Die Solaris 11.2 DTrace Dokumentation unter:
http://docs.oracle.com/cd/E36784_01/html/E36846/

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 wobei durch die begrenzte default Größe der Puffer durchaus ein Überlauf möglich ist. Abhilfe schafft hier das Setzen entsprechender Optionen auf der Kommandozeile oder im D-Skript.
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". Sie bietet Anwendern einen einfachen und meist vertrauten Zugang, wobei das *dtrace(1m)* Kommando das Compilieren von Skripten steuert und das Ergebnis in den Kernel lädt. Um dadurch die Stabilität des Systems nicht zu gefährden stellt "D" weder Sprungbefehle noch Schleifen-Konstrukte bereit. Darüber hinaus agiert *dtrace(1m)* gleichzeitig auch als *consumer* für die anfallenden Daten.

Als Datentypen stehen alle in "C" üblichen Typen zur Verfügung, allerdings gelegentlich unter anderen Bezeichnungen. Der Typ *uintptr_t* ist ein Beispiel hierfür. Neben gebräuchlichen arithmetischen und logischen Operationen glänzt "D" insbesondere durch die integrierte String-Verarbeitung, assoziative Arrays – vergleichbar mit *perl* Hashes – und vor allem Aggregationen. Letztere stellen sicherlich eines der mächtigsten DTrace Features dar, dazu später mehr.
Eine große Menge vordefinierter Variablen bietet einfachen Zugriff auf Prozess-IDs, credentials, Timer und vieles mehr. Hervorzuheben ist das *fds[]* array das die einem file-descriptor zugehörigen Daten (*fileinfo_t* Struktur) bereitstellt. Damit lässt sich auch bei *read()* oder *write()* Operationen sehr einfach die zugehörige Datei o.ä. ermitteln.

```
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;
```

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,* Nutzung von `-n pattern` schränkt diese bei Bedarf ein.

```
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 Lesezugriffe im System ist einfach mit dem folgenden Skript erreichbar:

```
obi-wan# cat reads.d

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

syscall::read:entry
{
    printf("%-16s %10s %s\n",
        execname,
        probefunc
        fds[arg0].fi_pathname
    );
}


obi-wan# ./reads.d
...
bacula-fd        read       /backup/mail/imap/l/user/PRIVACY/53065.
bacula-fd        read       /backup/mail/imap/l/user/PRIVACY/53065.
hwmgmtd          read       /tmp/hmptemp/ssmlibipmitool_stdout_Bnb3b
sshd             read       /devices/pseudo/clone@0:ptm
nscd             read       /etc/passwd
```

execname und *probefunc* sind Beispiele für 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 unterscheiden, in unserem Beispiel etwa *read(2)* und *readv(2)*.
Mit einer kleinen Anpassung lassen sich einzelne Anwendungen filtern:

```
syscall::read:entry
/ execname == $$1 /           # filter by name passed as argument
{
...
}

obi-wan# ./reads_mod.d hwmgmtd
hwmgmtd          read       /system/volatile/ssm-hwmgmtd.cache
hwmgmtd          read       /system/volatile/ssm-hwmgmtd.cache
hwmgmtd          read       /tmp/hmptemp/ssmlibipmitool_stdout5Cnb3b
...
```

`$1` wird durch das erste Kommandozeilen Argument, hier `hwmgmtd`, 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 die Lesbarkeit und das Parsen.

Auch die Zahl der empfangenen IP Pakete pro IP Adresse lässt sich ähnlich einfach ermitteln und dabei gleichzeitig eine weitere Stärke von DTrace demonstrieren, die Nutzung der Kommandozeile. Durch die effiziente Gestaltung lassen sich so die meisten Aufgaben mit Einzeilern 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
  5.150.153.110                   1
  134.60.1.72                    14
  134.60.112.134              19389
```

Aggregationen:
Aggregationen nutzen eine Besonderheit mancher mathematischer Funktionen aus. Für diese gilt, dass die Anwendung der Funktion auf eine Teilmenge der Daten im ersten Schritt und nachfolgend auf die Zwischenergebnisse im zweiten Schritt das selbe Ergebnis liefert 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 zwischenzuspeichern. Auch 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

In Solaris 11.2 neu hinzu gekommen:

- llquantice — log-lineare Verteilung

Insbesondere die jetzt drei Verteilungsfunktionen erweisen sich bei der Analyse von IO Problemen oft als hilfreich da hier Sprünge in den Größenordnungen der Messwerte, etwa der Latenz, besonders interessant sind.

Tipp:
Auch die Sortierung von Aggregationen bei der Ausgabe lässt sich mit DTrace Optionen steuern:

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

Folgende Optionen sind derzeit hierfür definiert:

- aggsortkey — Sortierung nach Index, nicht nach Wert wie voreingestellt
- 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 dienen soll durch setzen des Wertes für *aggsortpos* ausgewählt werden.

„IO“ Fragestellungen:
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 oft der bereits dargestellte Ansatz über die zugehörigen Systemaufrufe. Dieser läßt sich einfach um weitere Messwerte wie die Zahl übertragenen Bytes erweitern.

Sind jedoch auch die zugrunde liegenden Platten bzw. Geräte von Belang so bietet sich die Nutzung des *io-providers* an. Dieser stellt in seiner *start-probe* Strukturen bereit (vgl. C-struct), mit denen sich

sowohl die Namen der Dateien und Geräte als auch die Art der Operation, Schreiben bzw. Lesen, oder die Anzahl der zu übertragenden Bytes bestimmen 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 %10s\n", "DEVICE", "APP", "BYTES");
    printa("%10s %20s %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 einer Aggregation, hier bei Beendigung des Skriptes. Dazu findet die interne DTrace *probe* `END` Verwendung. Im Gegensatz zum ersten Beispiel erhalten wir hier also auf der logischen Ebene aggregierte Daten pro physikalischem Gerät.

```
obi-wan# ./file_io.d
^C
    DEVICE                       APP      BYTES
     sd109               sync_server      98304
     sd133               sync_server      98304
     sd141               sync_server      98304
     sd102               sync_server      99840
     sd102                zpool-data     178688
     sd118                zpool-data     178688
     sd134                zpool-data     178688
...
     sd125                 bacula-fd   11847168
     sd121                 bacula-fd   11858432
     sd128                 bacula-fd   12288000
     sd123                 bacula-fd   13051392
```

Offensichtlich besitzt das System einige Platten und ist mit Backup Aufgaben beschäftigt.

Frage 2: Welche Größenverteilung haben die Netzwerkpakete pro Anwendung und Verbindung?

Diese Frage lässt sich nur in Solaris 11 mit DTrace beantworten, da der notwendige *ip-provider* erst ab dieser Version verfügbar ist. 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/E36784_01/html/E36846/glhhr.html

Gut zu erkennen ist die Aufbereitung der Daten durch DTrace, beispielsweise die Bereitstellung von IP Adressen als Strings. Die Ausgabe von Aggregationen erledigt *dtrace(1m)* automatisch am Ende der Laufzeit des Skripts. Wie im Beispiel ersichtlich ist hierzu keine explizite Anweisung notwendig.

```
  134.60.84.147                                           sched
      value  ------------- Distribution ------------- count
         16 |                                         0
         32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3185
         64 |                                         6
        128 |                                         4
        256 |                                         5
        512 |                                         15
       1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1042
       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 eine 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
```

Mit derartigen Skripten lassen sich leicht die aktivsten NFS Klienten ermitteln, bei Bedarf inklusive der sich im Zugriff befindlichen Dateien.

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. Ein Fehler kann dazu führen das jegliche CPU Instruktion von DTrace „bearbeitet“ wird. Beispiele für „gute“ *probe* Definitionen 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 diesen Mitteln kann jetzt die Zeit gemessen werden, die eine Anwendung in einzelnen Funktionen, sowohl denjenigen der Anwendung als auch in Funktionen in externen Bibliotheken, zubringt. Um eine möglichst flexible Nutzung des Codes zu gewährleisten, werden auch hier die Namen der Bibliothek sowie der gewünschten Funktionen als Kommandozeilen Parameter übergeben. Wildcards, '*' und '?', können verwendet werden. Sie müssen jedoch unbedingt entsprechend maskiert werden, da sie sonst 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 einige memory-Routinen der *libc* beim Aufruf des *date* Kommandos ergibt sich dann:

```
obi-wan# dtrace -s x.d -c date 'libc' 'mem*'
Sunday, September 21, 2014 01:24:48 PM CEST
      2908ns    libc.so.1:membar_producer
      7717ns    libc.so.1:memset
      8716ns    libc.so.1:memcmp
     15177ns    libc.so.1:membar_consumer
     44359ns    libc.so.1:memmove
```

Hilfreich ist eine derartige Analyse insbesondere auch bei mit OpenMP parallelisierten Programmen um etwa Zeitverluste durch Synchronisation an Barrieren zu ermitteln.

```
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 diesen Techniken einfach darstellbar. DTrace unterstützt dies explizit durch die Kommandozeilen Option '-F'. Das folgende Beispiel verdeutlicht dies anhand des Aufrufes von Funktionen der C-Bibliothek *libc.so* durch das Kommando '*sleep 1*'.

```
#!/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 'sleep 1'
dtrace: script './libc_trace.d' matched 6293 probes
 CPU FUNCTION
   7  -> __tls_static_mods
   7    -> lmalloc
   7      -> getbucketnum
   7      <- getbucketnum
   7      -> initial_allocation
   7        -> __systemcall
   7        <- __systemcall
   7      <- initial_allocation
   7    <- lmalloc
...
```

Auch der Erweiterung eigener Anwendungen um *DTrace-provider* sind nahezu keine Grenzen gesetzt sofern der Quellcode zur Verfügung steht. Damit lassen sich die Standard-Funktionen um wertvolle Daten direkt aus der Anwendung ergänzen. Die notwendigen Hilfsmittel, um eigene *provider* in die Anwendung zu integrieren sind Bestandteil von Solaris, hier an einem einfachen C-Programm verdeutlicht. Dieses liest einzelne Zeichen und stellt deren ASCII Wert für DTrace via *provider* bereit.

```
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);
        /* more application code */
    }
}
```

Um die Anwendung "DTrace ready" zu machen, sind nur die rot markierten Ergänzungen notwendig:

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

Zusätzlich benötigt DTrace noch Informationen bzgl. der selber definierten *provider*. Diese finden sich unter http://docs.oracle.com/cd/E36784_01/html/E36846/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
```

Damit lässt sich nun eine ausführbare Anwendung erzeugen

```
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
```

und unter Kontrolle von DTrace starten. Im folgenden Beispiel verwenden wir der Einfachheit halber den bereits bekannten Quelltext der exemplarischen Anwendung als Eingabe.

```
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
```

„Backup“ Showcase:
Im Rahmen der Migration des an der Universität Ulm eingesetzten Backup Systems hin zur „Bacula Enterprise Edition“ traten im Rahmen der pre-production Tests größere Abweichungen hinsichtlich des erwarteten Durchsatzes auf. Das Betriebskonzept sieht folgenden Eckpunkte vor:

- Die anfallenden Backup-Daten aller Server werden immer zuerst auf Platte gesichert. Hierzu stehen zwei große ZFS-basierte Pools (RAIDZ2) mit einer Nettokapazität von jeweils ~150TB zur Verfügung.
- Die inkrementellen Backups verbleiben für Ihre gesamte Lebensdauer auf den Plattenpools.
- Full-Backups werden mit Ausnahme des initialen ausschließlich virtuell durch das Backup-System erzeugt was deutlich kürzere Laufzeiten ermöglicht und gleichzeitig die Server entlastet.
- Die full-Backups werden abhängig von Alter bzw. Füllgrad der Pools von Platte auf die verfügbaren Bandlaufwerke migriert und der Plattenbereich im Anschluss frei gegeben.

Die zu Grunde liegende Hardware und Software besteht aus:

- 2x Oracle SPARC-T4 Server mit 128GB Speicher
- 2x Quanta M4600H JBODs mit je 60 HGST 4TB SAS-2 Enterprise Platten
- 8x IBM 3592-E07 Fiber-Channel Bandlaufwerke
- PostgreSQL 9.2 auf 6x 800GB SAS-2 SSDs

Nach multi-stream IO-Messungen mit *filebench* ist sie in der Lage 3-5 GB/s, je nach Konfiguration, von den zpools zu lesen oder auf diese zu schreiben. Pro Bandlaufwerk wurden mit *btape*, einem zur Backup Software gehörenden Tool, 250-500 MB/s gemessen. Die Komprimierbarkeit der Daten durch das Laufwerk ist hierbei für die Schwankung verantwortlich.
Die Performance für die Sicherung eines durchschnittlichen Fileservers – 4,5 Millionen Dateien mit insgesamt ca. 6TB – blieb mit einer Migrationszeit von rund 30 Stunden und einer Datenrate von durchschnittlichen nur 61 MB/s weit hinter den Erwartungen zurück. Auch war der Zeitgewinn von nur 6 Stunden im Vergleich zum initialen Backup enttäuschend.

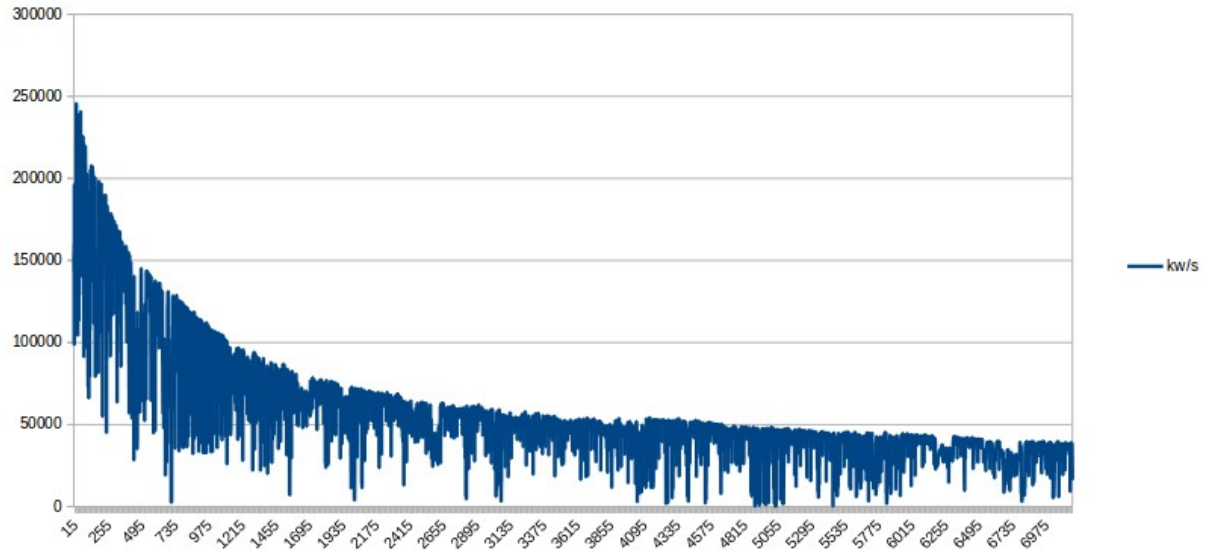

Abbildung 3: Tape IO-Performance

Eine Analyse mit Hilfe des DTrace *io-providers* lieferte Hinweise auf ein „schleichendes“ Problem, das erst mit längeren Laufzeiten auftritt. Abbildung 3 zeigt einen Ausschnitt der Messung der Disk-to-Tape IO-Bandbreite während der ersten 8 Stunden.

Ein sogenanntes hot-spot-sampling[4] des Kernels zeigt, dass hier keine CPU Engpässe vorliegen. Dabei werden 1001-mal pro Sekunde die *program-counter* der CPUs durch DTrace ausgelesen und den Kernel Routinen zugeordnet.

```
Routine                                 calls  prct.
SPARC-T4`copyin                          8630    0.2%
zfs`fletcher_4_native                   10902    0.3%
zfs`vdev_raidz_generate_parity_pq       19815    0.5%
zfs`lzjb_compress                      234676    5.9%
unix`cpu_halt                         3670636   91.8%
```

Ein entsprechendes sampling der Anwendung, gefolgt von einer tiefer gehenden DTrace Analyse der dort auffälligen Routinen lieferte den Entwicklern den passenden Hinweis auf mögliche Performance-Schwachstellen eines eingesetzten Algorithmus.
Eine uns nach wenigen Tagen bereit gestellte Testversion zeigte beeindruckende Verbesserungen. Mit einer Laufzeit von nur 8 Stunden ist die Migration 4-mal schneller und erreicht dabei durchschnittliche Datenraten jenseits der 200 MB/s inklusive ggf. notwendiger Bandwechsel.

Visualisierung mit DTrace:
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 4) schnell und übersichtlich darstellen.

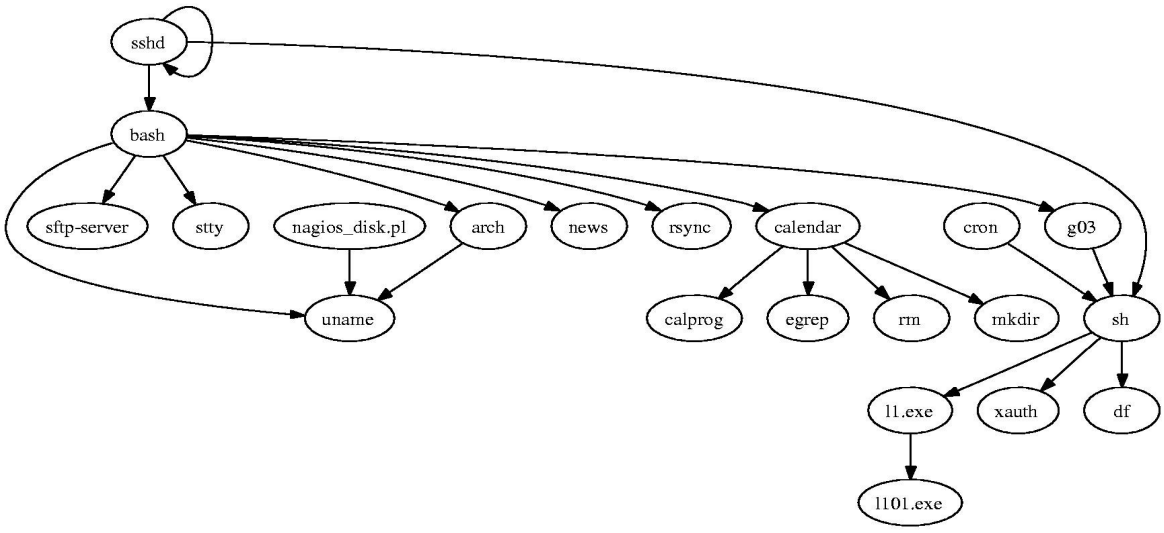

Abbildung 4: Prozesshierarchien

Der zu Grunde liegende Code ist – DTrace typisch – nur wenige Zeilen lang.

4 http://www.brendangregg.com/DTrace/hotkernel

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

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

Danksagung:
Mein besonderer Dank für die fortlaufende Unterstützung mit Anregungen, Ideen und Korrekturen gilt meinem Kollegen Dr. Harald Däubler. Weiterhin danke ich Arno Lehmann, Bacula Systems, für seine Unterstützung im Rahmen des „Backup Showcase“.

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