Analyse und Monitoring mit DTrace

Thomas Nau
Universität Ulm – kiz
Ulm

Schlüsselworte
DTrace, Performance, Analyse, Monitoring

Einleitung
Nicht nur durch die stark zunehmende Virtualisierung sind Echtzeitdaten hinsichtlich des Zustandes des Gesamtsystems unverzichtbar um die Performance von modernen Serversystemen bewerten und potentielle Engpässe pro-aktiv erkennen zu können. So können etwa die Wechselwirkungen zwischen den zahlreichen Gastsystemen mit ihren Anwendungen und dem zu Grunde liegenden Kernel oft Auslöser für Probleme im Massenspeicher- und Netzwerkumfeld sein.
Hoch skalierende Systeme – heute bereits oft mit Terabytes an Hauptspeicher und hunderten threads – erfordern ausgeklügelte Technologien um Anomalien, etwa von IO-Latenzen, zu erkennen. Diese sind sehr oft Auslöser von Performanceproblemen in jeglichem Anwendungsszenario.
Darüber hinaus wird der Trend zu Cloud basierten Lösungen und zu leichtgewichtiger Virtualisierung – etwa in Form von Solaris Zonen oder Docker – den Bedarf an entsprechenden Tools zur Analyse und zum Monitoring noch spürbar steigern.

Seit ihrem Erscheinen in Solaris 10 vor nun mehr als 12 Jahren setzt DTrace, die *dynamic tracing facility,* Maßstäbe im Bereich der System-Analyse und des Performance Monitoring und ist auch heute noch das Tool an dem sich die Werkzeuge anderer Betriebssysteme und Distributionen messen lassen müssen. Auf Linux Systemen hat der 2013 erstmals vorgeschlagene eBPF (*Enhanced Berkeley Packet Filter*) die Möglichkeit geschaffen einfach und mit einem Tool zu DTrace vergleichbare Ergebnisse zu erzielen.
Oracle baut DTrace im Solaris Umfeld aktiv als Grundlagen-Technologie weiter aus, setzt also andere Tools auf den DTrace Schnittstellen auf, und passt es an aktuelle Entwicklungen des Kernels an. DTrace ist ausser für FreeBSD auch für MacOS verfügbar.[1] Der Funktionsumfang reicht dabei nicht an den der Solaris Implementierung heran. Dies gilt auch für Linux Distributionen einschließlich Oracles *Unbreakable Linux*.

Dennoch gehen auch heute noch viele Systemadministratoren und Entwickler fälschlicherweise davon aus, dass DTrace ausschließlich ein Tool für Kernel-Hacker sei und nur von diesen zu bedienen und zu nutzen wäre. Sofern ein grundlegendes Verständnis über die Funktionsweise von UNIX Kerneln vorhanden ist, ist dem mitnichten so. Belegen lässt sich dies durch zahlreiche Beispiele aus den Bereichen Anwendungsentwicklung, Systemadministration aber auch Performance Analyse.
Einige dieser Beispiele werden auf den folgenden Seiten vorgestellt. Dabei wird sich zeigen, dass oft wenige Zeilen an D-Code, der Skriptsprache von DTrace, ausreichen um Informationen zu gewinnen die anderweitig nur schwierig oder gar nicht zugänglich wären. Auch auf die Anwendungsentwicklung hat die Betrachtung des Gesamtsystems positive Auswirkungen, da die ermittelten Performancedaten von höherer Qualität sind.

1 DTrace, Brendan Gregg, Jim Mauro, Prentice Hall, 2011

Hintergrund des Autors
Der Autor ist seit mehr als 30 Jahren im IT Umfeld tätig und leitet derzeit die „Abteilung Infrastruktur“ des Kommunikations- und Informationszentrums (kiz) an der Universität Ulm. Das kiz, dessen stellvertretender Leiter der Autor ebenfalls ist, trägt unter anderem die Gesamtverantwortung für die universitäre IT-Infrastruktur. Diese umfasst auch die 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. Des Weiteren ist die Abteilung sehr stark in Projekte[2] des Landes Baden-Württemberg eingebunden und erbringt in diesem Zusammenhang auch Dienstleistungen für weitere Hochschulen des Landes basierend auf dem leistungsfähigen Landeshochschulnetzes BelWü[3].

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

Statistische Tools, hierzu gehört zum Beispiel net*stat(1m)*, liefern lediglich statistische Kenngrößen, wie Mittelwerte über Zeitintervalle, die auf genauen Werten, wie etwa der Zahl der gesendeten Pakete, basieren. Dem Nachteil des Verlusts der Originalwerte steht der Vorteil eines schnellen jedoch groben Überblicks über den Zustand einzelner Kernel-Sub-Systeme gegenüber. Gravierend ist auch, dass eine Zuordnung kritischer Werte zu Anwendungen oder Nutzern nicht möglich ist. An dieser Stelle sei noch kstat(1m) erwähnt, ein sehr mächtiges aber wenig bekanntes Solaris Tool.

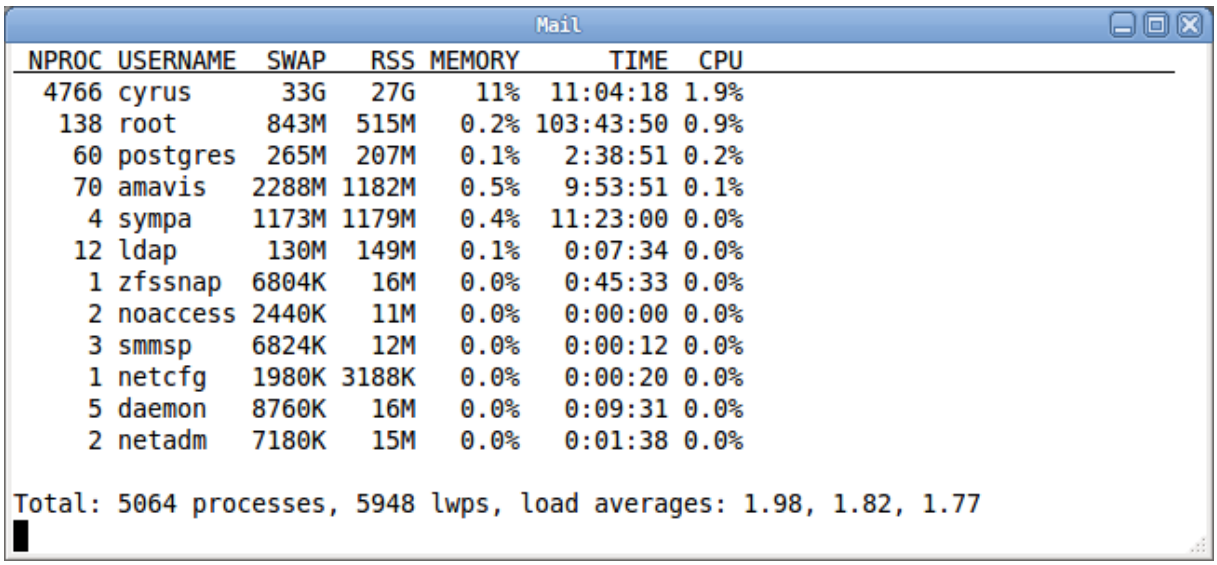

Abbildung 1: "prstat -t" Ausgabe

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

Sampling Tools, bekannt vertreten durch *prstat(1m)* und *top(1)*, unterliegen wie alle Mitglieder dieser Gruppe Einschränkungen, die denen des Nyquist-Shannon Abtasttheorems vergleichbar sind. So sind zum Beispiel kurzzeitige Spitzenlasten im Allgemeinen nicht erfassbar da die Samplingintervalle fast immer einige Größenordnungen über der Dauer dieser Spitzenlast Ereignisse liegen. Dies wird anhand der Diskrepanz zwischen aufaddierter CPU-Last von Prozessen und der Last des Systems – repräsentiert durch dessen *load* – in Abbildung 1 verdeutlicht. Auf der positiven Seite ermöglicht das gezielte sampling die Erfassung viele Parameter und deren Korrelation mit Anwendungen oder auch Nutzern.

Event basierte Tools, wie neben Debuggern und *truss(1)* insbesondere auch DTrace bzw. *eBPF* im Linux Umfeld, instrumentieren Anwendung und Kernel derart, dass die zu beobachtenden Ereignisse eine Funktion, meist das Sammeln oder die Auswertung zugehöriger Daten auslösen. Im Gegensatz zu t*russ(1)* oder *gdb(1)* beschränkt sich DTrace jedoch nicht auf die Anwendung nebst Bibliotheken und Systemaufrufen, sondern deckt auch den vollständigen Kernel und seine Module mit ab. Störend wirkt sich bei den alten Ansätzen auch die Tatsache aus, dass die Anwendung gestoppt wird um die Daten zu gewinnen. Dies kann erheblichen Einfluss auf das Timing haben und die Erkennung von transienten Fehlern bzw. Störungen unmöglich machen.

DTrace als 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.“*[4]

Durch seine dynamische Natur und Struktur sowie die Integration in den Kernel bietet DTrace für die vorgenannten Probleme in aller Regel eine Lösung mit Sicht auf das Gesamtsystem. Im Kern besteht DTrace hierbei aus nur wenigen Komponenten deren Nutzung oft auch ohne root-Privilegien möglich ist.

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

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

also zum Beispiel

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

Funktionell werden diese *probes* in sogenannte *provider* gruppiert. Deren wichtigste Aufgabe ist die Bereitstellung der Daten in aufbereiteter Form. So stellt z.B. der *ip-provider* Adressen als String und nicht in binärer Form zur Verfügung. Äußerst hilfrech sind die *provider,* die die Aufbereitung von protokollspezifischen Daten übernehmen. Hierfür existieren im aktuellen Solaris Implementierungen für *iSCSI, NFSv3, NFSv4, SMB, IP, UDP, TCP* und *SRP*. Unglücklicherweise ist der *SMB-provider*

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

nicht im DTrace Manual sondern im Soalris Handbuch[5] *Managing SMB File Sharing and Windows Interoperability in Oracle Solaris 11.3* zu finden.

Sogenannte *consumer*, der dritte wesentliche Bestandteil der Architektur, sind für das asynchrone Auslesen der durch die *provider* bereitgestellten Daten aus den Puffern des Kernels verantwortlich. Die Schnittstelle zum Solaris Kernel ist wie üblich über Bibliotheken (*libdtrace)* und Gerätetreiber implementiert.
Zu guter Letzt bedient sich DTrace einer eigenen Sprache Namens „D“, die an „C“ angelehnt ist und die meisten von dort bekannten Operatoren und Typen zur Verfügung stellt. Da „D-Code“ im Kernel ausgeführt wird, fehlen der Sprache jedoch aus Sicherheitsgründen Sprung- und Schleifenkonstrukte. Ergänzend stellt „D“ begrenzt String-Operationen und insbesondere assoziative Arrays zur Verfügung. Die lassen sich am besten mit *perl-hashes* vergleichen. D-Programme werden mit dem *dtrace(1m)* Kommando übersetzt und in den Kernel geladen. Gleichzeitig dient dieses Tool auch als *consumer* für die anfallenden Daten.
Im Gegensatz zu typischen Anwendungen hat ein D-Programm keinen Programmfluss, sondern seine Blöcke agieren als einzelne Routinen, die durch das Auslösen aktivierter *probes* aufgerufen werden, sofern die spezifizierten optionalen Bedingungen erfüllt sind.

Abbildung 2 verdeutlicht die Integration in den Solaris Kernel. Für weitergehende Informationen zu den Grundfunktionen sei an dieser Stelle lediglich auf die Artikel des Autors in den Tagungsbänden der *DOAG 2013* und *2014* sowie auf die Oracle Dokumentation unter

http://docs.oracle.com/cd/E36784_01/html/E36846

verwiesen.

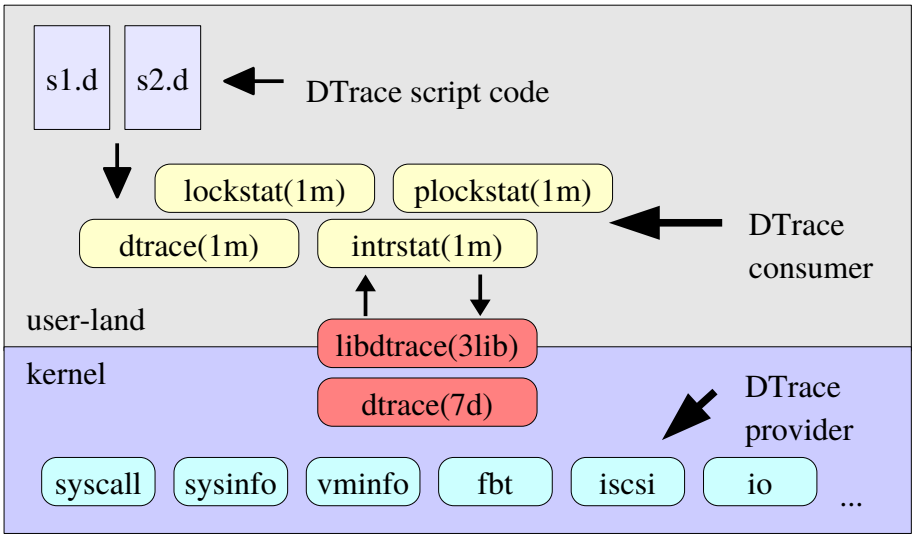

Abbildung 2: DTrace user-land / Kernel Schnittstelle

Aggregations, der DTrace Warp-Antrieb
Neben den von DTrace bereits vordefinierten Variablen und Strukturen ist vor allem der Einsatz von *aggregations* bei komplexeren Fragestellungen unabdingbar. Allen ist hierbei die nachfolgende mathematische Eigenschaft gemein: Wird die Funktion im ersten Schritt auf Teilmengen der Daten

5 https://docs.oracle.com/cd/E53394_01/html/E54797

angewandt und im zweiten Schritt auf diese Zwischenergebnisse so liefert dies identische Ergebnisse wie die Anwendung auf die Gesamtmenge aller Daten in einem Zug. Beispiele hierfür sind Minima und Maxima, aber auch die Summen-Funktion:

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

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

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

Tabelle 1 fasst die in Solaris 11.3 verfügbaren *aggregations* und ihre Funktion zusammen.

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

Tabelle 1: Aggregations in Solaris 11.3

Die Hilfsfunktionen zur Manipulation und zur Ausgabe von *aggregations* die von DTrace bereit gestellt werden fasst Tabelle 2 zusammen.

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

Tabelle 2: Hilfsfunktionen für aggregations

Die Ausgabe von in *aggregations* gespeicherten Daten geschieht mit Hilfe der *printa()* Funktion, die sich stark an *printf()* anlehnt jedoch automatisch über alle verwendeten Indizes iteriert und vorab eine Sortierung vornimmt. Wir erinnern uns, aus Sicherheitsgründen stellt „D“ keine Schleifenkonstrukte zur Verfügung. Der Format-String muss nicht zwingend für jeden Index ein Element enthalten jedoch

sind keine Lücken möglich. Der eigentliche Wert der *aggregation* wird durch den Format-String *%@* definiert, der auch mehrfach vorkommen kann. Die Ergebnisse der Quantisierungsfunktionen werden immer als ASCII-Grafik dargestellt. Hierzu folgen später Beispiele.

Hilfreich ist auch die Möglichkeit mehrere *aggregations* bei der Ausgabe über gemeinsame Indizes zu verknüpfen wie das folgende Code-Beispiel verdeutlicht, das einem einfachen *iostat(1m)* entspricht.

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

#pragma D option quiet

BEGIN {
    setopt("aggsortkey", "true");   /* sort by index (dev or app) */
    setopt("aggsortkeypos", "1");   /* sort by dev */
    ts = timestamp;
}

io:::start  /* trace physical read-IO */
/ args[0]->b_flags & B_READ /
{
    @r[execname, args[1]->dev_statname] = sum(args[0]->b_bcount);
}

io:::start  /* trace physical write-IO */
/ args[0]->b_flags & B_WRITE /
{
    @w[execname, args[1]->dev_statname] = sum(args[0]->b_bcount);
}

/* tick-provider fires on single CPU to trigger output of data;
 * makes use of first commandline argument to set rate
 */
tick-$1 {
        /* normalize to seconds and kilo-bytes
         * remember, dtrace times are nanoseconds
         */
    factor = 1024 * (timestamp - ts) / 1000000000;
    normalize(@r, factor);
    normalize(@w, factor);

        /* print headera and data
         * then reset counters and store new timestamp
         */
    printf("\n%-16s %-10s %16s %16s\n",
        "executable", "device", "read[kb/s]", "write[kb/s]");
    printa("%-16s %-10s %@16d %@16d\n", @r, @w);
    clear(@r);
    clear(@w);
    ts = timestamp;
}
```

Tipp: Da das Zeitintervall zusammen mit der Einheit an das Skript übergeben wird, kann diese nicht numerisch verarbeitet werden. Um dennoch einen Normalisierungsfaktor „pro Sekunde“ gewinnen zu können wird die vergangene Zeit über die DTrace Variable *timestamp* gemessen.

```
obi-wan# ./demo_iostat.d 5s
executable       device           read[kb/s]      write[kb/s]
zpool-rpool      sd189                     0               47
postgres         sd2                     762                0
postgres         sd254                  1335                0
postgres         sd255                  1213                0
zpool-rpool      sd4                       0               47
```

Wie im Beispiel gezeigt, lässt sich auch die Sortierung der Ausgabe steuern. Die hierfür notwendigen Optionen sind in der folgenden Tabelle 3 zusammengefasst.

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

Tabelle 3: DTrace Sortieroptionen

Wie eingangs erwähnt, spielen Latenzen bzw. deren Abweichungen von einem Normalwert immer öfter eine kritische Rolle für hoch skalierende Systeme und Anwendungen. Die Unterschiede in der zugrunde liegenden Hardware zeichnen sich in folgendem Beispiel deutlich ab.

```
jedi# ./demo_iolat.d 60s
R
           value  ------------- Distribution ------------- count
              32 |                                         0
              64 |                                         1
             128 |@@@@@@@@@@@@@@@@@@                       759
             256 |@@@@@                                    242
             512 |                                         2
            1024 |                                         0
            2048 |                                         2
            4096 |@                                        33
            8192 |@@@@@@@@@                                340
           16384 |@@@@@@                                   272
           32768 |@@                                       73
           65536 |@                                        25
          131072 |                                         8
          262144 |                                         5
          524288 |                                         4
         1048576 |                                         0
```

```
obi-wan# ./demo_iolat.d 60s
R
           value  ------------- Distribution ------------- count
              32 |                                         0
              64 |@@                                       2663
             128 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  41292
             256 |                                         117
             512 |                                         231
            1024 |                                         146
            2048 |                                         94
            4096 |                                         28
```

Lesezugriffe auf den ZFS 3-fach Spiegel aus Enterprise-SSDs von *obi-wan* laufen nahezu immer mit einer Latenz von 128-255µs ab. *Jedi* hingegen zeigt eine gänzlich andere Verteilung: einen deutlichen Peak bei ~200µs und einen unschärferen bei ~16ms. Die Erklärung liegt im Layout des *zpools*, der im Gegensatz zum vorgenannten aus herkömmlichen Festplatten aufgebaut ist denen ein SSD basierter L2ARC Cache zur Seite steht. Das zugehörige Skript basiert auf dem *io-provider*. Die Ausgabe ist der Übersichtlichkeit halber auf die Ausgabe auf Lesezugriffe beschränkt.

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

#pragma D option quiet

io:::start
{
        /* keep start time per device and block */
    start[args[0]->b_edev, args[0]->b_blkno] = timestamp;
}

    /* only handle IOs for which we have a start time */
io:::done
/start[args[0]->b_edev, args[0]->b_blkno]/
{
        /* use efficient "this" vaiables to set read/write flag
         * and to turn nanoseconds into microseconds
         */
    this->rwflag = args[0]->b_flags & B_WRITE ? "W" :
                   (args[0]->b_flags & B_READ  ? "R" : "*");
    this->us  = (timestamp - start[args[0]->b_edev, args[0]->b_blkno]);
    this->us /= 1000;
    @q[this->rwflag] = quantize(this->us);

        /* clear flag */
    start[args[0]->b_edev, args[0]->b_blkno] = 0;
}

/* tick-provider fires on single CPU to trigger output of data;
 * makes use of first commandline argument to set rate
 */
tick-$1 {
    printa("%@d\n", @q);
    clear(@q);
}
```

Erwähnenswert sind noch die möglicherweise problematischen Werte bei denen IO Operationen mehr als 0.5 Sekunden dauerten.

DTrace für Anwendungsentwickler

Der ggf. bisher entstandene Eindruck DTrace wäre nur ein Tool für Systemadministratoren täuscht. Eine grundlegende Analyse von Anwendungen ist möglich, in den allermeisten Fällen sogar ohne Zugriff auf den Quellcode.

Hierzu kommt der *pid-provider* zum Einsatz. Er hat eine Sonderstellung inne, da seine *probes* erst zur Laufzeit dynamisch generiert werden. Da die probes generell eine Überwachung bis auf die Ebene von Assemblerbefehlen erlauben, ist Vorsicht bei ihrer Definition geboten.

Über das DTrace Makro `$target` wird die Prozess-ID bereitgestellt. Sie basiert auf den beiden Kommandozeilenparametern `-c` bzw. `-p`. Möchte man beispielsweise die OpenMP „*spin*“ Funktionen analysieren verwendet man folgende probe:

```
pid$target:libmtsk.so.1:spin*:entry
```

Das folgende Beispiel zeigt diejenigen Funktionen der Bibliothek *libmtsk* auf, die hauptsächlich CPU-Zeit beanspruchen.

```
obi-wan# OMP_NUM_THREADS=2 ./lib_timing.d -c './partest 100 10' libmtsk ''
...
Timing total, top-10 only
 time [us]              module:function
      3837       libmtsk.so.1:thread_cancel_point
      3923       libmtsk.so.1:getfsr
      3959       libmtsk.so.1:push_context
      4431       libmtsk.so.1:barrier_reset_nthreads
      4532       libmtsk.so.1:getpsr
      4693       libmtsk.so.1:package_a_task
      5122       libmtsk.so.1:pop_context
      7433       libmtsk.so.1:_omp_affinity_mode
     35567       libmtsk.so.1:ready_to_work
     60326       libmtsk.so.1:sleep_at_barrier
```

Die im zugehörigen DTrace-Skript verwendete Variable *vtimestamp* stellt die Zeit bereit, die ein Thread auf einer CPU aktiv war. Damit lässt sich aus der letzten Zeile ableiten, dass die OpenMP Implementierung hier wohl „*spin-wait*“ an Stelle von „*thread-sleep*“ zur Synchronisation verwendet. Dies lässt sich zur Laufzeit durch Umgebungsvariable steuern, jedoch sei an dieser Stelle lediglich auf die OpenMP und Compiler Dokumentation verwiesen.

Betrachtet man die *aggregation,* die die CPU-Zeit auf einzelne Threads der Anwendung aufschlüsselt, so lassen sich leicht Ungleichgewichte in der Lastverteilung erkennen. Deutlich ist im Beispiel die stark unterschiedliche Verteilung der CPU-Zeit auf die 4 Threads im Verhältnis 44:30:19:7 erkennbar. In diesem Fall ist dies jedoch dem der Anwendung zugrunde liegenden Algorithmus geschuldet. Meinen Dank an Dieter an Mey von der RWTH Aachen für die Bereitstellung der Fortran-90 Codes.

```
obi-wan# OMP_NUM_THREADS=4 ./lib_timing.d -c ./transpose transpose ''
...
Timing per thread, top-10 only
 time [us]   TID       module:function
          5     1    transpose:_start
        269     1    transpose:init_misaligned_data_trap_handler
     459892     1    transpose:_$d1A17.MAIN_
     462289     4    transpose:_$d1A17.MAIN_
     464534     3    transpose:_$d1A17.MAIN_
     466945     2    transpose:_$d1A17.MAIN_
    1515176     4    transpose:_$d1B27.MAIN_
    4265414     3    transpose:_$d1B27.MAIN_
    6830205     2    transpose:_$d1B27.MAIN_
    9938128     1    transpose:_$d1B27.MAIN_
```

Das zugehörige Analyse-Skript:

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

#pragma D option quiet

pid$target:$1:$2:entry {
    self->ts = vtimestamp;
}

pid$target:$1:$2:return
/ self->ts /
{
    self->delta = vtimestamp -self->ts;
    @total[probemod, probefunc]       = sum(self->delta);
    @thread[tid, probemod, probefunc] = sum(self->delta);
    self->ts = 0;
}

END {
    normalize(@thread, 1000);
    trunc(@thread, 10);
    printf("Timing per thread, top-10 only\n");
    printf("%10s %5s %12s:%s\n",
        "time [us]", "TID", "module", "function");
    printa("%10@d %5d %12s:%s\n", @thread);

    trunc(@total, 10);
    normalize(@total, 1000);
    printf("\n");
    printf("Timing total, top-10 only\n");
    printf("%10s       %12s:%s\n",
        "time [us]", "module", "function");
    printa("%10@d       %12s:%s\n", @total);
}
```

Ungleichgewichte spielen jedoch nicht nur bei Verteilung der Threads auf CPU-Cores eine Rolle, sondern insbesondere auch bei Speicherzugriffen. Bei heutigen Systemen mit zwei und mehr Sockeln

ist NUMA (Non-Uniform Memory Access) quasi inhärent. Für speicherintensive Anwendungen kann dies zu einer nicht deterministischen Performance führen, da die Zugriffsgeschwindigkeit teils stark von der Lokalität zwischen Thread und verwendetem Speicherbereich abhängt. Viele Optimierungen sind ebenso Bestandteil des Solaris Kernels, wie die Möglichkeit für Anwendungen dem Kernel mittels *madvise(3c)* Hinweise zu geben. Für einige Gruppen von Anwendungen, etwa im Bereich der Graphenverarbeitung, ist die Speicherperformance oft mit entscheidend. Die dort oft auftretenden random-access Muster stellen erhebliche Anforderungen an die Speicheranbindung, aber auch an die Umsetzungseinheiten zwischen virtuellen und physikalischen Speicheradressen etwa in Form des TLB (translation look-aside buffer). Die vergleichsweise geringe Zunahme der Speichergeschwindigkeit der letzten Jahre im Vergleich zur CPU-Entwicklung fordert hier ihren Tribut.

Die Analyse dieser Probleme ist hochgradig plattformspezifisch und erfordert weitgehendes Wissen über die CPU- und System Architektur, besonders bezüglich der Interpretation der von den Hardware-Countern gelieferten Werte. Einen ersten Eindruck über die Komplexität liefern:

> http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/120214-t4-d04-hp-ext-performance-2307329.pdf
>
> http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/120214-t4-d04-hp-ext-performance-2307329.pdf

Seit einigen Jahren bietet der *cpc-provider* in Solaris eine enge Integration mit DTrace und eine sehr gute Alternative zum Einsatz von *cputrack(1)* und *cpustat(1m)*. Hierbei wird das generische[6] PAPI (Performance Application Programming Interface) vom cpc-*provider* ebenso unterstützt wie die plattformspezifischen Zähler. Das folgende Skript listet die Anwendungen, die auf einem SPARC-T4 System für die „kostspieligen“ DTLB-Traps verantwortlich sind. Die DTrace *probe* löst für je 10.000 Ereignisse einmal aus.

```
T4# dtrace -n 'cpc:::DTLB_fill_trap-all-10000 { @[execname] = count(); }'
dtrace: description ' cpc:::DTLB_fill_trap-all-10000 ' matched 1 probe
^C

  sshd                                                              2
  ldmd                                                              6
  gzip                                                             27
  zpool-ssd                                                        35
  postgres                                                         36
  sched                                                            70
  tar                                                             106
```

Bleibt zu erwähnen, dass diese Werte für den Messzeitraum weniger Sekunden vollkommen unkritisch sind.

Ablaufanalyse

Besonders hilfreich sind auch die *proc-* und *sched-provider,* die die Aspekte der Prozess- und Thread-Erzeugung bzw. deren CPU scheduling abdecken. Sie ermöglichen die Verfolgung von Threads bei der Migration zwischen Cores oder, hinsichtlich der Speicherzugriffe interessanter, zwischen locality-groups[7]. Diese Details sprengen jedoch den Umfang des Artikels.

6 siehe auch *generic_events(3CPC)*

Das nachfolgende Beispiel soll dem Leser dennoch die Mächtigkeit der o.g. *provider* demonstrieren und auch zeigen, wie gut sich die Stärken von DTrace, das Sammeln relevanter Daten, mit der Stärke externer Tools, hier zur Visualisierung, verbinden lassen.

Das Folgende mit Hilfe des *proc-providers* und *dot* aus dem *GraphViz* Paket erstellte Diagramm verdeutlicht die *fork()/exec()* Zusammenhänge beim Aufruf von `man ls` aus einer Shell.

```
digraph ExecPaths {
  "bash-2206"  -> "bash-2241";
  "ksh93-2244" -> "col-2244";
  "ksh93-2243" -> "nroff-2243";
  "man-2242"   -> "ksh93-2242";
  "ksh93-2242" -> "ksh93-2243";
  "ksh93-2242" -> "ksh93-2244";
  "man-2245"   -> "ksh93-2245";
  "ksh93-2245" -> "mv-2245";
  "man-2246"   -> "ksh93-2246";
  "ksh93-2246" -> "less-2246";
  "man-2241"   -> "man-2242";
  "man-2241"   -> "man-2245";
  "man-2241"   -> "man-2246";
  "bash-2241"  -> "man-2241";
}
```

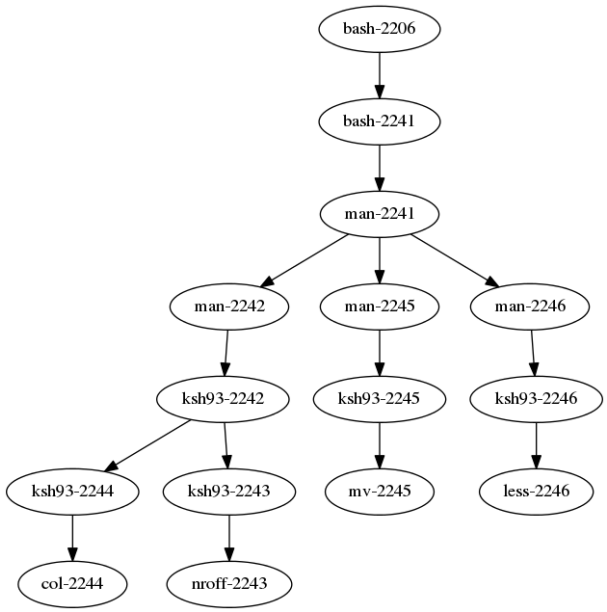

Abbildung 3: fork()/exec() Hierarchie "man ls"

7 Aus *plgrp(1)*: An lgroup represents the set of CPU and memory-like hardware devices that are at most some distance (latency) apart from each other.

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

#pragma D option quiet

BEGIN
{
        /* trace only childs of the passed in pid */
    trflag[$target] = 1;
    printf("digraph ExecPaths {\n");
}

proc:::create
/ trflag[pid] /
{
        /* trace the new child pid after a fork()
         * else we break the chain; execname stays the same
         */
    trflag[args[0]->pr_pid] = 1;
    printf("  \"%s-%d\" -> \"%s-%d\";\n",
        execname, pid, execname, args[0]->pr_pid);
}

proc:::exec
/ trflag[pid] /
{
        /* need to keep the parents name/pid as the exec
         * may still fail and the data isn't available
         * anymore if it succeeds
         */
    self->parent = execname;
    self->ppid   = pid;
}

proc:::exec-success
/ self->parent != NULL /
{
    printf("  \"%s-%d\" -> \"%s-%d\";\n",
        self->parent, self->ppid, execname, pid);
    self->parent = 0;
    self->ppid   = 0;
}

proc:::exit
/ trflag[pid] /
{
    trflag[pid] = 0;
}

END
{
    printf("}\n");
}
```

Bring your own code
Sofern Zugriff auf die Quellen einer Anwendung besteht, lassen sich auch eigene spezifische *provider* einfach in diese integrieren. Der Vorteil gegenüber dem tracing von Anwendungsfunktionen liegt, wie bei *providern* üblich, in der Aufbereitung der interessanten Daten.
Für die häufig genutzten Anwendungen MySQL[8] sowie den Apache Webserver[9] bringt Solaris 11.3 bereits die entsprechende Unterstützung mit.
Das folgende Beispiel zeigt wie einfach DTrace mit einem C-Programm zu kombinieren ist. Das eigentliche Programm hat in diesem Fall beispielhaft lediglich die Aufgabe einzelne Zeichen von *stdin* zu lesen. Diese werden dann einer eigens definierten *probe* als Argument übergeben. Die mittels

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

eingebundene Datei stellt die notwendigen Makros zur Verfügung, die eine einfache Übergabe an DTrace ermöglichen.

```
#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 */
    }
}
```

Zusätzlich wird eine Datei – hier `keypress_d.d` – benötigt, die die Prototypen der *probes* unseres *providers* namens *keypress* definiert.

```
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
```

Nun müssen die beiden Teile übersetzt und gelinkt werden. Es ist darauf zu achten ob es sich um eine 32-bit, wie in unserem Fall, oder 64-bit Anwendung handelt.

```
nau@obi-wan:/test> cc -O -c keypress.c
nau@obi-wan:/test> dtrace -32 -G -s keypress_d.d keypress.o
nau@obi-wan:/test> cc -O -o keypress keypress_d.o keypress.o -ldtrace
```

Gestartet von der Kommandozeile ergibt sich folgende Ausgabe:

8 https://dev.mysql.com/doc/refman/5.7/en/dba-dtrace-server.html
9 https://github.com/oracle/solaris-userland/tree/master/components/apache2-modules/mod_dtrace

```
cat keypress.c keypress_d.d |
    dtrace -c ./keypress \
           -n 'keypress$target:::read { @ = lquantize(arg0,0,255,32); }'
dtrace: description 'keypress$target:::read ' matched 1 probe
dtrace: pid 17443 has exited


           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@                                       31
              32 |@@@@@@@                                  111
              64 |@@                                       37
              96 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  421
             128 |                                         0
```

Unser Programm *keypress* wird mit Hilfe der Kommandozeilenoption `-c` von DTrace gestartet. Damit steht die *PID* dem ebenfalls über die Kommandozeile spezifizierten DTrace code als `$target` zur Verfügung. Letztendlich wird dann der Integer-Wert jedes einzelnen Zeichens in Form einer linearen Quantisierung in einer *aggregation* gespeichert.
Die „Weite“ der einzelnen Quantisierungsschritte beträgt 32. Daher liefert das Histogramm einen Hinweis auf die Verteilung der Zeichen hinsichtlich Groß- und Kleinschreibung.

Speculations, ein weiteres DTrace power-feature
Ein Problem mit dem insbesondere Entwickler oft konfrontiert sind, sind Fehler die scheinbar zufällig und sporadisch auftreten. Der Einsatz eines Debuggers kann sich schwierig gestalten und die beliebten *printf()* statements im Code setzen entweder eine Vermutung über die Auslöser voraus oder sie produzieren Unmengen zusätzlicher an sich wertloser Information. Beispiele für solche Fehler sind Funktionen die z.B. nach 10.000 Aufrufen einen Fehler liefern oder, aus Kernel-Sicht, Systemaufrufe und IO-Operationen die ungewöhnlich lange dauern. In beiden Fällen kann erst nach erfolgter Abarbeitung der Routinen entschieden werden ob diese erfolgreich oder nicht durchgeführt wurden. Für die Analyse notwendige Daten sind zu diesem Zeitpunkt jedoch oft nicht mehr verfügbar.
DTrace *speculations* adressieren dieses Problem indem sie „spekulativ“ Daten sammeln, sie aber nur im Bedarfsfall bereitstellen und ansonsten verwerfen. Das nachfolgende Beispiel, die Ausgabe wurde für diesen Druck entsprechend formatiert, gibt einen Einblick in die Möglichkeiten. Die Fragestellung ist die Identifizierung von erfolgreichen Schreiboperationen deren Dauer den auf der Kommandozeile übergebenen Schwellwert, hier 10ms, überschreiten.

Beim Aufruf von *write()* werden die nur dann verfügbaren Daten „Filedeskriptor“, „Dateiname“, und „Größe“ zusammen mit einem Zeitstempel in einer neu erstellten Thread-spezifischen *speculation* abgelegt. Die Filedeskriptoren *stdin*, *stdout* und *stderr* werden ausgeblendet. Für ein stärker belastetes System ist die Zahl der Spekulationen ggf. durch Setzen der DTRace Variable *nspec* zu erhöhen.

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

#pragma D option quiet
#pragma D option nspec=16

dtrace:::BEGIN
{
    us_fac = 1000;
    limit  = $1 *us_fac;
}

syscall::write:entry
/ arg0 >= 3 /
{
    self->spec = speculation();
    self->ts   = timestamp;
    speculate(self->spec);
    printf("%-9s %6d bytes FD %2d (%s):  ",
        execname, arg2, arg0, fds[arg0].fi_pathname);
}

syscall::write:return
/ self->spec && (timestamp -self->ts) >= limit && arg0 != -1 /
{
    speculate(self->spec);
    printf("%dus\n", (timestamp -self->ts) / us_fac);
    commit(self->spec);
    self->spec = 0;
}

syscall::write:return
/ self->spec /
{
    discard(self->spec);
    self->spec = 0;
}
```

Das Ergebnis:

```
jedi# ./demo_spec.d 10000
imapd       96 bytes FD 17 (/mail/imap/.../cyrus.index):           167131us
lmtpd     1512 bytes FD 20 (/mail/imap/.../cyrus.cache):            33896us
postgres  8192 bytes FD 10 (/mail/postgres/data/base/16386/16418):  12605us
lmtpd     1360 bytes FD 20 (/mail/imap/.../DNS-Output/cyrus.cache): 24313us
lmtpd     1360 bytes FD 20 (/mail/imap/.../Mizar/cyrus.cache):      28209us
sendmail  5120 bytes FD 23 (<unknown>):                             33377us
postgres 16384 bytes FD 30 (/mail/postgres/data/pg_xlog/...):       78915us
```

Destructive actions

Einige DTrace Operationen können erheblichen Einfluss, insbesondere negativen, auf Prozesse oder das Gesamtsystem haben. Diese sind unter dem Oberbegriff *destructive actions* zusammengefasst und müssen explizit aktiviert werden. Dies kann sowohl durch die Kommandozeilenoption `-w` oder durch

ein `D pragma` erreicht werden. Für eine vollständige Übersicht sei auf die DTrace Dokumentation[10] verwiesen.
In der Vergangenheit hat es sich oft als besonders hilfreich erwiesen, Rückgabewerte des Kernels für bestimmte Anwendungen zu modifizieren. Dies können Speichergröße oder die Größe des swap-Bereiches sein aus denen *installer* gelegentlich unpassende Parameter ableiten. Die Informationen die *uname(1)* liefert sollen hier als Beispiel dienen.
Im Rahmen des Solaris Platinum-Beta Programms werden am kiz der Universität Ulm seit geraumer Zeit auch Solaris 12 Systeme mit großem Erfolg betrieben. Die aufgetretenen Anwendungsprobleme sind dabei meist Code geschuldet, der explizite Solaris Versionen bei der Installation voraussetzt und hierbei für Fehler sorgt wie der nachfolgende Auszug aus einem *preinstall* Skript zeigt.

```
case `uname -r` in
    5.10)
        /usr/sbin/svccfg -v import $file
        ;;

    5.11)
        svcadm restart manifest-import
        ;;

    *)
        echo "SMF Manifest not available on Solaris version `uname -r`"
        exit 1
        ;;
esac
```

Die aufwändige Lösung besteht in der Korrektur der Skripte, was jedoch mit Binärdateien unmöglich wäre. In letzterem Fall könnten dann allenfalls mit `LD_PRELOAD` einige Bibliotheksroutinen durch angepasste ersetzt werden. Weitaus einfacher ist es jedoch, mit DTrace die notwendigen Werte on-the-fly zu manipulieren wie das nachfolgende Beispiel demonstriert.

Beim Aufruf wird der Wert des ersten Argumentes gesichert, der Zeiger auf den Speicherbereich, der mit Daten gefüllt werden soll. Nachdem der Kernel seine Arbeit getan hat werden die notwendigen Teile davon verändert. Hier die Version und der Name des Systems. *copyin()* und *copyout()* übertragen Daten zwischen dem Adressbereich des Kernels, hier läuft DTrace, und dem der Anwendung in dem die Daten abgelegt werden sollen. *copyout()* zählt zu den *destructive actions.*

10 https://docs.oracle.com/cd/E36784_01/html/E36846/gkwvl.html

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

struct utsname  uts;

syscall::uname:entry
{
        self->buf = arg0;
}

syscall::uname:return
/ self->buf /
{
    this->p = (struct utsname *)
        copyin(self->buf, sizeof(struct utsname));
    bcopy("neo",  this->p->nodename, sizeof(uts.nodename));
    bcopy("5.11", this->p->release,  sizeof(uts.release));
    copyout(this->p, self->buf, sizeof(struct utsname));
    self->buf = 0;
}
```

Angewandt:

```
trinity# uname -a
SunOS trinity 5.12 s12_82 i86pc i386 i86pc

trinity# ./demo_uname.d
dtrace: script './demo_uname.d' matched 2 probes
dtrace: could not enable tracing: Destructive actions not allowed


trinity# ./demo_uname.d -w &
[1] 8630
dtrace: script './demo_uname.d' matched 2 probes
dtrace: allowing destructive actions

trinity:.../~# uname -a
SunOS neo 5.11 s12_82 i86pc i386 i86pc
```

Zusammenfassung

Durch seinen dynamischen Ansatz in Verbindung mit der großen Anzahl verfügbarer *provider* und *probes* bietet DTrace vielfältige Möglichkeiten zur Lösung von Problemen, die mit herkömmlichen Tools allenfalls mit sehr hohem Aufwand in den Griff zu bekommen wären. Selbst mit wenigen Zeilen D-Code lassen sich oft komplexe Analysen erstellen. Erwähnenswert ist unter diesem Gesichtspunkt besonders das DTrace Toolkit welches in der Zwischenzeit Bestandteil von Solaris ist. Die Suche nach schwer reproduzierbaren Fehlern wird erst dadurch ermöglicht, dass sich sowohl Anwendungssoftware als auch Kernel vielfältig instrumentieren lassen und damit eine umfassende Gewinnung von Debug- und Monitoring-Daten möglich wird.

Danksagung:
Mein besonderer Dank für die fortlaufende Unterstützung mit Anregungen, Ideen und Korrekturen gilt meinem Kollegen Dr. Harald Däubler sowie meiner Frau für die Spende zahlreicher Kommas und die Beseitigung sprachlicher Ungereimtheiten.

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