

Looking Into The Black-Box- how the kernel may impact your application



Thomas Nau

Thomas.Nau@uni-ulm.de

kiz

University of Ulm

Looking Into The Black-Box- how the kernel may impact your application



**Some scaring before
we get started ...**

Looking Into The Black-Box-

how the kernel may impact your application



Know your enemy and know yourself; in a hundred battles, you will never be defeated.

(Sun Tzu, about 2500 years ago)

Know your systems and your tools; for a hundred problems, you will always know what to do.

(adapted for today's IT centric world)

The “Old” Way

- *truss(1)*, *apptrace(1)* and *mdb(1)*
 - not dynamic; problems with timing or transient errors
 - probing affects the target
 - *truss* stops the process, gathers data and then continues the process
 - hard to combine data from different processes such as dtlogin session
 - hard to cross userland-kernel boundary with one tool

The “Old” Way

- *intrstat(1m)*
 - gathers and displays run-time interrupt statistics per device and processor or processor set

```
obi-wan# intrstat 5
```

device	cpu0	%tim	cpu1	%tim	cpu2	%tim	cpu3	%tim
e1000g#0	0	0.0	3899	7.6	0	0.0	0	0.0
e1000g#1	4067	8.1	0	0.0	0	0.0	0	0.0
uhci#0	1	0.0	0	0.0	0	0.0	0	0.0
uhci#1	1	0.0	0	0.0	0	0.0	0	0.0

```
^C
```

of interrupts percentage of absolute time spent in IRQ handler

Looking Into The Black-Box-

how the kernel may impact your application



The “Old” Way

- *busstat(1m)*
 - report bus-related performance statistic
 - hardware support required (output depends on system)

```
yedi# busstat -w dram0,pic0=bank_busy_stalls,pic1=mem_read_write 2
```

time	dev	event0	pic0	event1	pic1
2	dram0	bank_busy_stalls	193307539	mem_read_write	71400764
4	dram0	bank_busy_stalls	192952912	mem_read_write	71338904
6	dram0	bank_busy_stalls	194118606	mem_read_write	71866659
8	dram0	bank_busy_stalls	194180462	mem_read_write	71822108
10	dram0	bank_busy_stalls	129238477	mem_read_write	49908669
12	dram0	bank_busy_stalls	1029510	mem_read_write	641610
14	dram0	bank_busy_stalls	10229	mem_read_write	11562

The “Old” Way

- *cputrack(1m), cpustat(1m)*
 - monitor system- and/or application performance using CPU hardware counters

```
yedi# cputrack -v -t -c pic0=DTLB_miss,pic1=Instr_cnt -p 24450
```

time	lwp	event	%tick	pic0	pic1
0.008	1	init_lwp	0	0	0
1.014	1	tick	21508400	2583	5222045
2.014	1	tick	3477900	624	595049
3.014	1	tick	72532	0	0

^C

The “Old” Way

- *trapstat(1m)*
 - reports trap statistic per processor or processor set
 - most useful for TLB related events as they can have a severe impact on performance
 - compare large TLB caches of UltraSPARC-IIIi versus the tiny ones found in new UltraSPARC-T1 based T2000 systems
 - had significant bad effect on our web-mail application before we tuned it using large pages

Looking Into The Black-Box- how the kernel may impact your application



The “Old” Way

```
yedi# trapstat -c 4 -T
```

cpu	m	size	itlb-miss	%tim	...	dtlb-miss	%tim	dtsb-miss	%tim	%tim
4	u	8k	70	0.0	...	1031	0.0	2	0.0	0.0
4	u	64k	130	0.0	...	898	0.0	13	0.0	0.0
4	u	4m	0	0.0	...	1	0.0	0	0.0	0.0
4	u	256m	0	0.0	...	0	0.0	0	0.0	0.0
-----+-----+-----+-----+-----										
4	k	8k	172	0.0	...	39162	1.8	28	0.0	1.8
4	k	64k	0	0.0	...	0	0.0	0	0.0	0.0
4	k	4m	0	0.0	...	5785	0.3	69	0.0	0.3
4	k	256m	0	0.0	...	0	0.0	0	0.0	0.0
=====+=====+=====+=====+=====										
	tt1		372	0.0	...	46877	2.2	112	0.0	2.2

Looking Into The Black-Box-

how the kernel may impact your application



The “Old” Way

- the **stat(1m)* commands are very helpful but generally provide top-level views only
- most available 3rd party tools are userland centric
- **hint:** *user_attr(4)* can be used to grant non-root users access to hardware performance counters

```
nau:::defaultpriv=basic,cpc_cpu
```



Preparing For A New Era

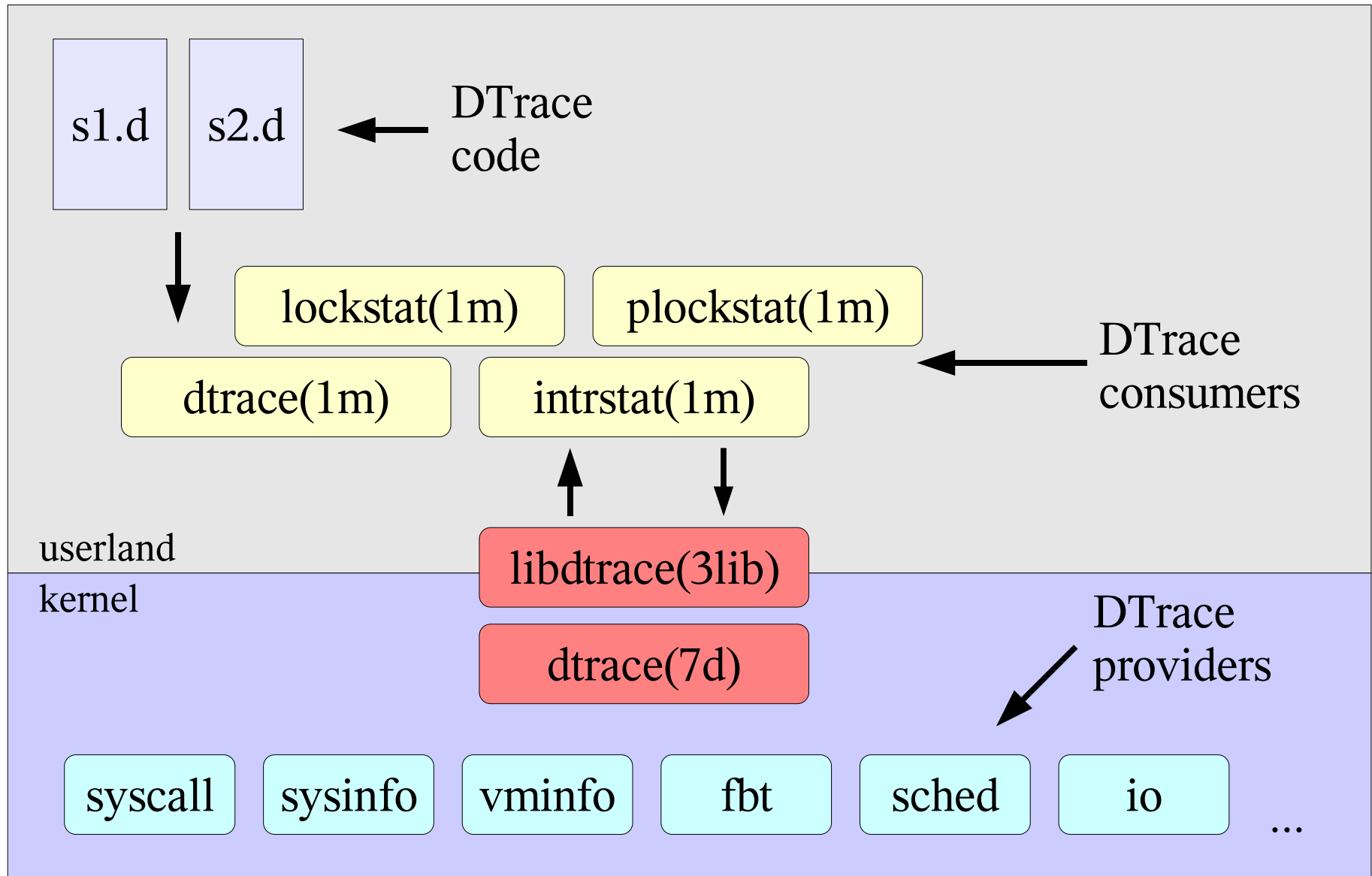
- solution to the mentioned problems
 - dynamically modifying a system in a safe way to record arbitrary data
 - replace sampling by triggers
- in other words ...



DTrace

- dynamic tracing facility
 - allows to dynamically and efficiently instrument kernel and user-level code
 - 40,000+ probes distributed over kernel and modules; can be enabled independently with almost no overhead
 - C-like scripting, interpreted in kernel context (no loops)
 - tools already built on top: *plockstat(1m)*, *er_kernel(1)*, ...
 - access based on *Least Privileges*; you may grant user *dtrace_{proc, user, kernel}* privileges

Looking Into The Black-Box- how the kernel may impact your application



DTrace Probes

- are locations or activities which can trigger almost arbitrary actions
 - record user/kernel stacks
 - printout data structures
 - even manipulate data
- identified by provider, module, function and name
 - *sched:unix:resume:off-cpu*

DTrace Build-In Probes

- *dtrace::: {BEGIN, END, ERROR}*
 - trigger when DTrace script starts, finishes or when an error occurs

```
obi-wan# dtrace -n 'dtrace:::BEGIN { trace("Here We Go"); } \  
                  dtrace:::END   { trace("We Are Done"); }'
```

```
dtrace: description 'dtrace:::BEGIN ' matched 2 probes
```

CPU	ID	FUNCTION:NAME	
2	1	:BEGIN	Here We Go
^C			
2	2	:END	We Are Done

```
obi-wan#
```

DTrace Providers

- DTrace probes are implemented by so named *providers*
- each of them performs a particular type of probing
 - *syscall*: offers probes for every entry and return point of all system calls
 - *fbt*: implements probes for entering and leaving kernel functions
- some providers create probes on-the-fly
- you may also write your own

“D-Language” Structure

- no *loops* or similar control statements for safety reasons as DTrace code is executed in kernel context (just think about an endless loop)
- very simple general form executed top to bottom; predicates are optional

```
probe description
/ predicate /
{
    actions
}
```

Example: System Call Statistics

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

/* use syscall provider and count() aggregation
 * to create system call 'call-statistics'
 */

syscall:::entry
/ execname != "dtrace" /
{
    @c[execname, probefunc] = count();
}
```



Example: System Call Statistics

```
obi-wan# ./scstat.d
^C
...
Xsun          writev          23
Xsun          pollsys         27
gnome-terminal pollsys         28
Xsun          read           30
gnome-terminal ioctl          35
rpcbind       lwp_sigmask     36
ypserv        lwp_sigmask     36
rpcbind       fstat           45
rpcbind       ioctl          54
java          pollsys         55
Xvnc          pollsys        162
```

DTrace Script Language “D”

- “C” like look and feel
- provides scalars, strings, associative arrays, structs and unions, pointers and access to kernel variables (e.g. ``kmem_flags``)
- includes basic arithmetic, logical and relational expressions
- built-in variables
 - `pid`, `ppid`, `tid`, `cpu`, `pset`, `probenname`, `probefunc`, ...

The Good Stuff: Aggregations

- aggregations have an outstanding property
 - applying a function to a subset of the data and again to the achieved results gives the same result as applying the aggregation to the whole dataset
 - SUM is an aggregating function MEDIAN is not
- helps a lot to condense data
 - no need to keep the complete dataset
 - no scaling problems
- *printa()* might be used to format the printout

Example: *write()* Timing

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

/* example take from /usr/demo/dtrace/writetimeq.d */

syscall::write:entry
{
    self->ts = timestamp;
}

syscall::write:return
/ self->ts /
{
    @time[execname] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

power-of-two distribution



More On DTrace Output

- *trace()* prints collected data in a predefined way
- *tracemem()* copies some memory junk to the DTrace buffer
- *stack()* and *ustack()* dump kernel- or user stack
hint: both can be used as aggregation “index”
- *printf()*, *printa()* work similar to the “C” version with some extensions
- aggregations are printed by default when exiting

Example: I/O Observations

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

#pragma D option quiet

io:::start {
    @c[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", @c);
}
```

Looking Into The Black-Box-

how the kernel may impact your application



Example: I/O Observations

```
obi-wan# ./fileio.d  
^C
```

DEVICE	FILE	APP	BYTES
ssd6	<none>	faker	1130496
nfs12	489.dat	sched	32768
nfs12	478.dx	sched	32768
ssd0	cis_Pd_complex.chk	1502.exe	5980160
ssd2	cis_Pd_complex.chk	1502.exe	6356992
ssd2	pt_slab.DM	siesta-constr	7946240
ssd0	pt_slab.DM	siesta-constr	7995392
md10	cis_Pd_complex.chk	1502.exe	12337152
md10	pt_slab.DM	siesta-constr	15941632

ssd0	Gau-3413.rwf	1502.exe	16588800
ssd2	Gau-3413.rwf	1502.exe	17686528
md10	Gau-3413.rwf	1502.exe	34275328

SVM mirrored volume

The *pid* Provider

- the *pid* provider allows you to trace any function or instruction in a user process
- probes are created on demand and will therefore not appear in the “*dtrace -l*” output
- instructions are specified as function offset
 - *pid54321:my-object:my-function:8*
- works by default similar to the *fbt* (function boundary provider) which is for the kernel level

Example: Library Call Stacks

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

/* uses the 'pid' provider to print call stacks */

#pragma D option quiet
                                commandline arguments
pid$target: $1:$2:entry
{
    printf("%s:%s:%s", probeprov, probemod, probefunc);
    ustack();
}
```

Example: Library Call Stacks

```
obi-wan# ./piddemo1.d -c ls 'libc' 'str*'
pidtest.d ←————— output of “ls”
```

```
pid1002:libc.so.1:strcmp
        libc.so.1`strcmp
        libc.so.1`setlocale+0x1378
        ls`main+0x22
        ls`_start+0x7a
```

```
pid1002:libc.so.1:strlen
        libc.so.1`strlen
        ls`xstrdup+0x12
        ls`main+0x486
        ls`_start+0x7a
```

```
pid1002:libc.so.1:strlen
        libc.so.1`strlen
        ls`xstrdup+0x12
```

Example: Library Timing (Inclusive)

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

/* uses the 'pid' provider to examine library timing */

#pragma D option quiet

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

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

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

Example: Library Timing (Inclusive)

```
obi-wan# OMP_NUM_THREADS=2 \  
./piddemo2.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
```



Detailed *sched/proc* Example

- DTrace code uses the *sched* and *proc* provider to observe thread scheduling and preempting
- simple test application consisting of nested loops compiled with *-xautopar -xloopinfo*
- resulting code runs in parallel
- demonstrated DTrace techniques also apply to any kind of local parallel application such as OpenMP based ones

Looking Into The Black-Box- how the kernel may impact your application



**Some details omitted,
please check handout**

The *sched* Provider

- makes probes related to CPU scheduling available
- easy way to examine why threads sleep, run or change priority
- allows you to keep an eye on thread migration critical for OpenMP or other local (SMP level) parallel applications

The *sched* Provider Probes

- on-cpu fires when a thread begun execution
- off-cpu fires when it's about to end execution
- preempt thread will be preempted
- remain-cpu dispatcher elected to continue thread
- sleep sleeping on synchronization object
- wakeup current thread wakes a sleeping one



The *proc* Provider

- it's probes are related to
 - process creation and termination
 - LWP creation and termination
 - *exec(2)*, *fork(2)*
 - signal handling



The *proc* Provider Probes

- `exec-failure` fires when `exec(2)` failed
- `exec-success` when `exec(2)` succeeded
- `exit` when process is exiting
- `lwp-create` on LWP creation
- `lwp-start` before first instruction is executed
- `lwp-exit` current LWP is exiting
- `start` before first instruction is executed

Detailed *sched/proc* Example

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    long    i, j, *a, *b;
    long    iter = 100000 *atoi(argv[1]),
           rep  =   100 *atoi(argv[2]);

    a = malloc(iter * sizeof(long));
    b = malloc(iter * sizeof(long));
    for (i = 0; i < iter; i++)
        a[i] = b[i] = i;
    puts("LOOP");
    for (j = 0; j < rep; j++)
        for (i = 0; i < iter; i++)
            a[i] *= b[i];
}
```

Detailed *sched/proc* Example

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

```
/* traces threads in application to observe scheduling  
 * behaviour with respect to CPUs and locality groups  
 *  
 * timing is done in micro seconds  
 */
```

```
#pragma D option quiet
```

```
/* initialize timestamp */
```

```
BEGIN
```

```
{
```

```
    baseline = walltimestamp;
```

```
    scale     = 1000; /* convert nano- to microseconds */
```

```
}
```

Detailed *sched/proc* Example

```
/* sched:::on-cpu fires whenever a thread
 * starts executing on a CPU
 * need to handle three different cases:
 * - init, called just once for a given thread
 * - thread has been migrated from another CPU
 * - "catch everything else" clause
 */
sched:::on-cpu init case detection
/ pid == $target && !self->stamp / {
    self->stamp = walltimestamp;
    self->lastcpu = curcpu->cpu_id;
    self->lastlgrp = curcpu->cpu_lgrp;
    stamp = (walltimestamp -baseline) / scale;
    printf("TID=%-2d %9d:%-9d CPU %3d(%d) created\n",
        tid, stamp, 0, curcpu->cpu_id, curcpu->cpu_lgrp);
}
```

Detailed *sched/proc* Example

```
/* clause 2: thread CPU migration
 */
sched:::on-cpu
/ pid == $target && self->stamp &&
  self->lastcpu != curcpu->cpu_id /
{
    delta = (walltimestamp -self->stamp) /scale;
    self->stamp = walltimestamp;
    stamp = (walltimestamp -baseline) / scale;
    printf("TID=%-2d %9d:%-9d from-CPU %d(%d)
           to-CPU %d(%d) migration\n",
           tid, stamp, delta, self->lastcpu, self->lastlgrp,
           curcpu->cpu_id, curcpu->cpu_lgrp);
    self->lastcpu = curcpu->cpu_id;
    self->lastlgrp = curcpu->cpu_lgrp;
}
```

Detailed *sched/proc* Example

```
/* clause 3: catch-all other sched:::on-cpu events
 */
sched:::on-cpu
/ pid == $target && self->stamp &&
  self->lastcpu == curcpu->cpu_id /
{
    delta = (walltimestamp -self->stamp) /scale;
    self->stamp = walltimestamp;
    stamp = (walltimestamp -baseline) / scale;
    printf("TID=%-2d %9d:%-9d CPU %3d(%d)
           restarted on same CPU\n",
           tid, stamp, delta,
           curcpu->cpu_id, curcpu->cpu_lgrp);
}
```

Detailed *sched/proc* Example

```
/* fires just before thread get's kicked-off a CPU
 */
sched:::off-cpu
/ pid == $target && self->stamp /
{
    delta = (walltimestamp -self->stamp) /scale;
    self->stamp = walltimestamp;
    stamp = (walltimestamp -baseline) / scale;
    printf("TID=%-2d %9d:%-9d CPU %3d(%d)
        taken from CPU\n",
        tid, stamp, delta,
        curcpu->cpu_id, curcpu->cpu_lgrp);
}
```

Looking Into The Black-Box- how the kernel may impact your application



Detailed *sched/proc* Example

```
/* fires when a thread is put asleep
 */
sched:::sleep
/ pid == $target && self->stamp /
{
    self->sobj = (
        curlwpsinfo->pr_stype == SOBJ_MUTEX      ? "kernel mutex" :
        curlwpsinfo->pr_stype == SOBJ_RWLOCK     ? "kernel RW lock" :
        curlwpsinfo->pr_stype == SOBJ_CV        ? "cond var" :
        curlwpsinfo->pr_stype == SOBJ_SEMA      ? "kernel semaphore" :
        curlwpsinfo->pr_stype == SOBJ_USER      ? "user-level lock" :
        curlwpsinfo->pr_stype == SOBJ_USER_PI   ? "user-level PI lock" :
        curlwpsinfo->pr_stype == SOBJ_SHUTTLE   ? "shuttle" : "unknown");
    delta = (walltimestamp -self->stamp) /scale;
    self->stamp = walltimestamp;
    stamp = (walltimestamp -baseline) / scale;
    printf("TID=%-2d %9d:%-9d sleeping on '%s'\n",
        tid, stamp, delta, self->sobj);
}
```

Detailed *sched/proc* Example

```
/* fires when LWP exits
 */
proc:::lwp-exit
/ pid == $target && self->stamp /
{
    stamp = (walltimestamp -baseline) / scale;
    printf("TID=%-2d %9d:%-9d exited\n",
           tid, stamp, 0);
}
```

Detailed *sched/proc* Example

- output shows
 - thread ID
 - relative time for thread referring to application start
 - relative time referring to last event triggered by thread
 - CPU number and locality group
 - informational message
- be aware that output for several threads does not need to be in order as it's buffered and may come from different processors

Looking Into The Black-Box- how the kernel may impact your application



Detailed *sched/proc* Example

```
obi-wan# ./threadsched.d -c './partest 10 10'
```

```
TID=1          0:0          CPU  25(0)  created
TID=1          0:0          CPU  25(0)  restarted on same CPU
TID=1          0:0          CPU  25(0)  taken from CPU
TID=1          0:0          CPU  25(0)  restarted on same CPU
TID=1      10004:10004      CPU  25(0)  taken from CPU
LOOP
TID=1      69999:59995      CPU  25(0)  restarted on same CPU
TID=1      120000:50000      CPU  25(0)  taken from CPU
TID=1      130003:10002      CPU  25(0)  restarted on same CPU
TID=1      6640553:6510550    CPU  25(0)  taken from CPU
TID=1      6640553:0          from-CPU 25(0) to-CPU 30(0) migration
TID=1      6640553:0          CPU  30(0)  restarted on same CPU
TID=1      10070842:0          from-CPU 30(0) to-CPU 2(0)  migration
TID=1      10070842:0          CPU   2(0)  restarted on same CPU
TID=1      10070842:3430289    CPU  30(0)  taken from CPU  miss-ordered timing
```

Enhancing the *proc/sched* Example

```
/* fires when a thread is put asleep
 * print call stack if thread sleeps on a
 * condition variable or user-level lock
 */
sched:::sleep
/ pid == $target &&
    ( curlwpsinfo->pr_stype == SOBJ_CV ||
      curlwpsinfo->pr_stype == SOBJ_USER ||
      curlwpsinfo->pr_stype == SOBJ_USER_PI) /
{
    ustack();
}
```

Looking Into The Black-Box- how the kernel may impact your application



Detailed *sched/proc* Example

```
TID=1          190004:0          CPU  24(0) taken from CPU
TID=1          210007:20002       CPU  24(0) restarted on same CPU
LOOP
TID=2          6110504:5980503   sleeping on 'cond var'
                libc.so.1`__lwp_park+0x10
                libc.so.1`cond_wait_queue+0x28
                libc.so.1`cond_wait+0x10
                libc.so.1`pthread_cond_wait+0x8
                libmtdk.so.1`sleep_at_barrier+0x6c
                libmtdk.so.1`__mt_EndOfTask_Barrier_+0xb8
                partest`_$_d1B15.main
                libc.so.1`_lwp_start

TID=2          6110504:0          CPU  16(0) taken from CPU
```

Looking Into The Black-Box- how the kernel may impact your application



Detailed *sched/proc* Example

```
# simulating overcommitment: 3 threads 2 CPU processor set
```

```
obi-wan# OMP_NUM_THREADS=3 ./threadsched.d -c './partest 10 10' |  
grep migration | sort -n -k 2,2
```

```
TID=1      60000:60000      from-CPU 4 (0) to-CPU 0 (0) migration  
TID=2      180009:50003     from-CPU 4 (0) to-CPU 0 (0) migration  
TID=1      530037:0         from-CPU 0 (0) to-CPU 4 (0) migration  
TID=1      630045:0         from-CPU 4 (0) to-CPU 0 (0) migration  
TID=2      730054:100008   from-CPU 0 (0) to-CPU 4 (0) migration  
TID=3      730054:0         from-CPU 4 (0) to-CPU 0 (0) migration  
TID=1      830062:100008   from-CPU 0 (0) to-CPU 4 (0) migration  
TID=2      830062:0         from-CPU 4 (0) to-CPU 0 (0) migration  
TID=1      930071:0         from-CPU 4 (0) to-CPU 0 (0) migration  
TID=3      930071:100008   from-CPU 0 (0) to-CPU 4 (0) migration  
TID=1      1550123:0        from-CPU 0 (0) to-CPU 4 (0) migration  
TID=1      1650131:0        from-CPU 4 (0) to-CPU 0 (0) migration  
TID=2      1730137:80006   from-CPU 0 (0) to-CPU 4 (0) migration
```

Visualizing DTrace Output

- DTrace has no built-in visualization options but can create easy to parse output
- use *perl(1)* to postprocess data
- tools like *dot*, which is part of the Graphviz package, greatly help visualizing DTrace output
 - *<http://www.graphviz.org/>*
- *gnuplot* is another option

Graphviz Example: Execution Paths

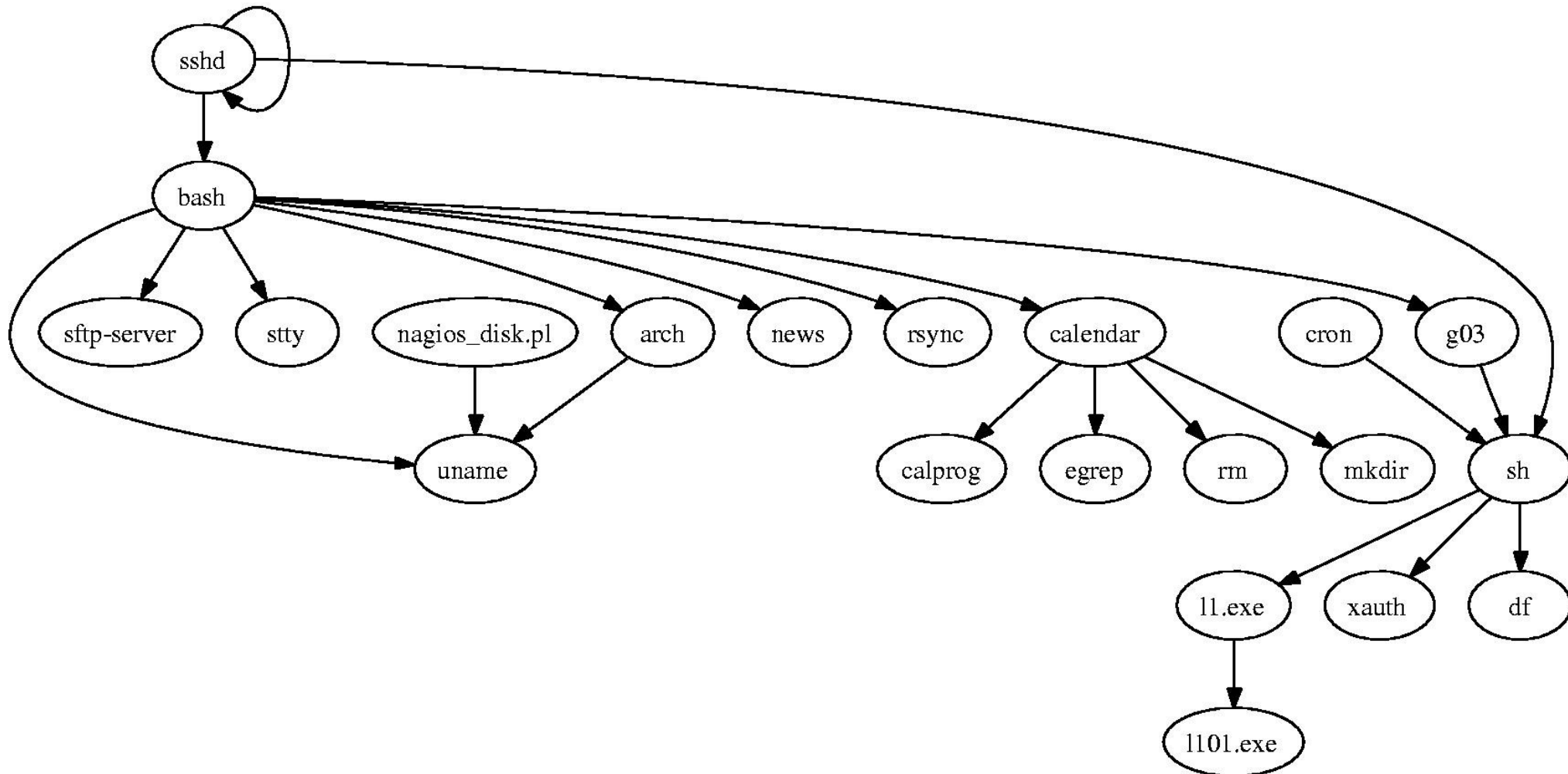
```
#!/usr/sbin/dtrace -s

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");
}
```

Graphviz Example: Execution Paths



Visualization Using SunStudio

- *er_kernel(1)* creates an experiment using data gathered by DTrace in the Solaris kernel
 - *er_kernel dd if=/dev/zero of=/dev/null *
bs=1024k count=10000
- results can be visualized either using *er_print(1)* or *analyzer(1)*

Looking Into The Black-Box- how the kernel may impact your application



The screenshot shows the Performance Analyzer interface for the application 'ktest.1.er'. The main window is divided into several panes. The top pane shows a table of functions with their KCPU cycles. The bottom pane shows a detailed view of the selected function 'syscall_trap32', including its PC address, size, source file, and process times.

Name	KCPU Cycles (sec.)	KCPU Cycles (sec.)	KCPU Cycles (sec.)
<Total>	7,015	23,817	23,817
syscall_trap32	0,010	0,010	7,015
read	6,615	0,	6,615
pollsys	0,110	0,010	0,110
cstatat64_32	0,040	0,	0,040
ioctl	0,040	0,	0,040
write	0,040	0,	0,040
exece	0,030	0,	0,040
copen	0,020	0,	0,020
cstatat32	0,020	0,	0,020
cfork	0,010	0,	0,010
door_call	0,010	0,	0,010
door_return	0,010	0,	0,010

Data for Selected Object:

Name: syscall_trap32
PC Address: 3:0x00041D8C
Size: 412
Source File: (unknown)
Object File: ktest.1.er/archives/unix
Load Object: <unix>
Mangled Name:
Aliases:

Process Times (sec.) / Counts

Exclusive

KCPU Cycles:	0,010 (42016,81%)	7
" count:	10027014000000	702893

DTrace Destructive Actions

- a number of DTrace actions may change the state of a system
- these actions need to be explicitly enabled by “-w” or “destructive” pragma
- process destructive actions
 - *stop()*, *raise()*, *copyout()*, *copyoutstr()*, *system()*
- kernel destructive actions
 - *breakpoint()*, *panic()*, *chill()*

Example: Time Warp

```
/* fake uname() syscall return values */

struct utsname  uts;

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

syscall::uname:return {
    p = (struct utsname *) copyin(self->buf,
        sizeof(struct utsname));
    bcopy("OpenSunOS", p->sysname,  sizeof(uts.sysname));
    bcopy("leia",      p->nodename,  sizeof(uts.nodename));
    bcopy("5.12",     p->release,   sizeof(uts.release));
    copyout(p, self->buf, sizeof(struct utsname));
}
```

process destructive action

Example: Time Warp

prompt set by shell, not by kernel



```
obi-wan# uname -a  
SunOS obi-wan 5.10 Generic_118833-33 sun4v sparc SUNW,Sun-Fire-T200
```

```
obi-wan# dtrace -w -s timewarp.d &
```

“-w” required because
destructive actions
are used

```
obi-wan# uname -a  
OpenSunOS leia 5.12 Generic_118833-33 sun4v sparc SUNW,Sun-Fire-T200
```

Speculations

- outstanding DTrace debugging feature
- great to catch sporadic or nondeterministic error conditions
- allow you to gather data and decide some time *after* certain probes have fired if the trace should be committed or discarded
- codepath leading to IO operations which take “extremely” long may serve as an example

Spice Up Your Own Applications

- DTrace allows you to easily add providers and probes to your own code

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

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

    while ((c = getchar()) != EOF) {
        if (c == 10) continue;
        DTRACE_PROBE1(keypress, read, c);
    }
}
```

Spice Up Your Own Applications

- definition is easy and straight forward
- *dtrace(1m)* is later used to “compile” the code

```
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
```

Spice Up Your Own Applications

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

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

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

obi-wan# dtrace -c ./keypress -n \
    'keypress$target:::read { @[arg0] = sum(1);' < file

dtrace: description 'keypress$target:::read ' matched 1 probe
dtrace: pid 18828 exited with status 1

          99                6
         100                6
          97                70
```

Looking Into The Black-Box- how the kernel may impact your application



DTrace Wrap-up

Some More Answers

- Do I need to recompile my code?
 - No, you don't have to. DTrace technology can be applied without having access to the sources but of course having symbols not removed from the executable helps
 - Recompiling allows you to add your own providers.
- Do I need to be root to make use the tool?
 - No, administrators can grant users secure access to the DTrace facility using privileges in *user_attr(4)*

Starting Points

- my favorite still is the *Solaris Dynamic Tracing Guide* which can be downloaded as PDF file from <http://docs.sun.com>
- great examples can be found at
 - <http://www.brendangregg.com/dtrace.html>
 - <http://www.opensolaris.org>
 - </usr/demo/dtrace>
 - <http://www.solarisinternals.com/si/dtrace/>

Looking Into The Black-Box-

how the kernel may impact your application



Summary

- DTrace is an outstanding and awful powerful system analysis tool; a *Swiss Army Knife* useful to administrators as well as developers
- even more powerful if used with speculation feature or spiced-up own code



Case Study #1

Tool Usage On T2000

Integer Performance

“john” Benchmark

- during T2000 tests we were looking at “john”
- password “evaluation” tool
- seemed to be the ideal candidate for quick'n dirty benchmarking using it's DES benchmark feature
 - small memory footprint
 - no floating point operation

Processors

- T2000, 1GHz UltraSPARC T1
 - 8 cores, 4 threads each using round-robin method on active ones; single instruction issue in-order design
 - 3MB unified 12-way L2 cache with 4 banks
 - 64 entry each I-TLB and D-TLB cache per core
- V100, 550MHz UltraSPARC-IIe
 - single core with 4-way superscalar pipeline
 - 512kB unified 4-way L2 cache
 - 64 entry each I-TLB and D-TLB cache

First Results

- used compiler flags *-fast -xarch=native64*

```
nau@v100:~/john-1.6.40/run> ./john --test --format:DES  
Benchmarking: Traditional DES [64/64 BS]... DONE  
Many salts:      230157 c/s real, 271314 c/s virtual  
Only one salt:  231421 c/s real, 234701 c/s virtual
```

```
nau@t2000:~/john-1.6.40/run> ./john --test --format:DES  
Benchmarking: Traditional DES [64/64 BS]... DONE  
Many salts:      159731 c/s real, 159731 c/s virtual  
Only one salt:  136724 c/s real, 136724 c/s virtual
```

Bad Performance

- comparing clock rates we expected number to be in the 400000-500000 range
- quick check on a 1GHz V240 (UltraSPARC-IIIi) proves the assumption could be right

```
nau@v240:~/john-1.6.40/run> ./john --test --format:DES
Benchmarking: Traditional DES [64/64 BS]... DONE
Many salts:      497999 c/s real, 497999 c/s virtual
Only one salt:   432274 c/s real, 433139 c/s virtual
```

Using *cputrack(1m)*

- *cputrack* gets us number of instructions per cycle

```
nau@t2000:~/john-1.6.40/run> cputrack -o profile -T 5 -t \  
-c pic1=Instr_cnt ./john --test -format:DES
```

```
nau@t2000:~/john-1.6.40/run> cat profile
```

time	lwp	event	%tick	pic1	
5.019	1	tick	5013226812	3401744911	
10.029	1	tick	6412314672	3182470158	
10.032	1	exit	11426217732	6584425110	<-- ratio 0.58

```
nau@v100:~/john-1.6.40/run> cat profile
```

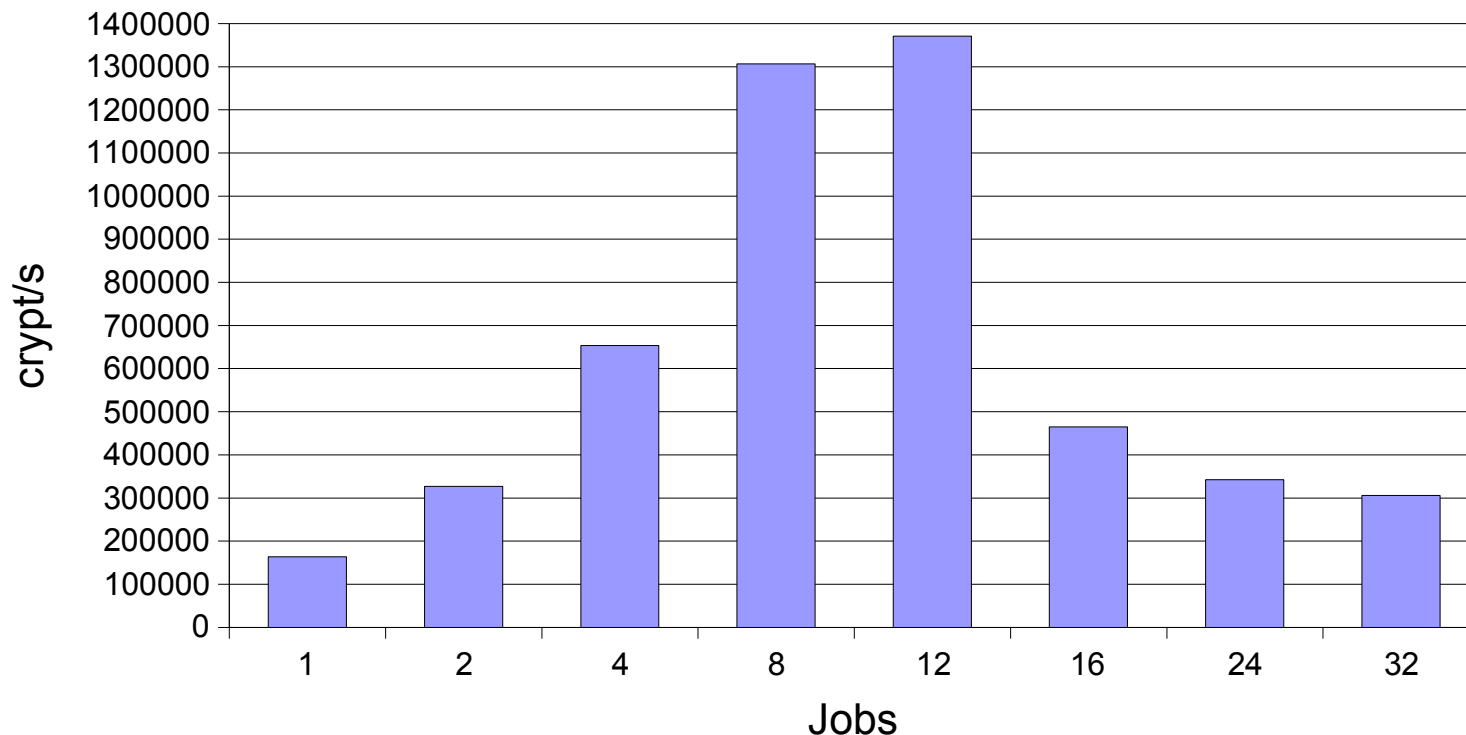
time	lwp	event	%tick	pic1	
5.021	1	tick	2700326309	5720338749	
10.021	1	tick	2664938937	5298458427	
10.064	1	exit	5386690436	11060175783	<-- ratio 2.05

First “john” Problem Solved

- unexpected “bad” performance can be explained by looking at the CPU design
- single in-order issue pipeline of T1 versus 4-way superscalar one of the Ite processor
- lessons learned:
 - forget about clock rates
 - the design matters

What About Scaling?

- system considered to be designed for throughput



Unexpected Scaling

- scaling drops rapidly for more than 12 job;
idea: related to 12-way L2 cache?
- changed Makefile to add hardware counter support for Sun Studio 11 *analyzer* and *collect*:
-g -xhwcprof
- added T1 specific hints to *analyzer* rc-file
(got them from Sun)
 - provide additional memory and cache information

Looking Into The Black-Box- how the kernel may impact your application



Performance Analyzer [bad.1.er]

File View Timeline Help

Find Text:

Summary Event Legend

User CPU (sec.)	CPU (%)	Wall (sec.)	User CPU (sec.)	Name	Max. Mem. Stall (sec.)	Max. Mem. Stall (sec.)
10.037	100.00	10.057	10.037	<Total>	9.387	9.387
9.897	98.60	9.917	9.897	DES_bs_crypt_25	9.266	9.266
0.090	0.90	0.090	0.090	DES_bs_set_key	0.080	0.080
0.040	0.40	0.040	0.040	DES_bs_expand_keys	0.030	0.030
0.010	0.10	0.010	0.010	DES_bs_set_salt	0.010	0.010
0.	0.	0.	10.037	_start	0.	9.387
0.	0.	0.	0.090	bench_set_keys	0.	0.080
0.	0.	0.	9.767	benchmark_all	0.	9.126
0.	0.	0.	10.027	benchmark_format	0.	9.377
0.	0.	0.	0.040	crypt_all	0.	0.030
0.	0.	0.	0.040	fmt_self_test	0.	0.020
0.	0.	0.	10.037	main	0.	9.387

Load Object: <john>

Original Name:

Aliases:

Process Times (sec.) / CPU

	Exclusive
User CPU:	9.897 (98.60%)
Wall:	9.917 (98.61%)
Total LWP:	9.917 (98.61%)
System CPU:	0. (0.%)
Wait CPU:	0.020 (100.00%)
User Lock:	0. (0.%)
Page Fault:	0. (0.%)
Page Fault:	0. (0.%)
Other Wait:	0. (0.%)
Mem. Stall:	9.266 (98.72%)

Unexpected Scaling

- memory stalls seem to be the major problem
- almost 99% are caused by a single routine
 - DES_bs_crypt_25
- let's have a look at hardware counter data related to the memory/cache subsystem

Looking Into The Black-Box- how the kernel may impact your application



The screenshot shows the Performance Analyzer interface. The left pane displays a table of memory objects with their maximum memory stall times. The right pane shows detailed data for the selected object, UST1_Bank Memory Object 3, including its process times and a circled maximum memory stall value of 4.993 seconds (53.20%).

Name	Max. Mem. Stall (sec.)
<Total>	9.387
UST1_Bank Memory Object 0	1.511
UST1_Bank Memory Object 1	1.561
UST1_Bank Memory Object 2	1.321
UST1_Bank Memory Object 3	4.993

Data for Selected Object:
Memory Objects: UST1_Bank Memory Object 3

Process Times (sec.) / Counts
Data-derived
Max. Mem. Stall: 4.993 (53.20%)

Looking Into The Black-Box- how the kernel may impact your application



The screenshot shows the Performance Analyzer interface. The left pane displays a list of memory objects under the 'Vpage_64K' category. The right pane shows the 'Summary' tab for the selected object, displaying 'Max. Mem. Stall: 8.026 (85.50%)' circled in red.

Performance Analyzer [bad.1.er]

File View Timeline Help

Find Text:

Display Mode: Text Graphical

Name
<Total>
64K Virtual Page 65558 (0x100160000)
64K Virtual Page 65557 (0x100150000)
64K Virtual Page 281474976677887 (0xffffffff7fff0000)
64K Virtual Page 65556 (0x100140000)

Data for Selected Object:

Memory Objects: 64K Virtual Page 65558 (0x100160)

Process Times (sec.) / Counts

Max. Mem. Stall: 8.026 (85.50%)

Memory Bank Hotspot

- one memory bank causes half of all stalls, even more, 85%, are caused by a single page
- Solaris uses larger pages for sun4v architecture to help small TLB caches; this may create hot spots as it limits cache mapping flexibility
- select a different cache-bin allocation algorithm (coloring) in */etc/system*
 - set `consistent_coloring=2`
- use *mdb(1m)* to experiment with live system

Looking Into The Black-Box- how the kernel may impact your application



Checking With *busstat(1m)*

```
yedi# busstat -w dram0,pic0=bank_busy_stalls,\  
pic1=mem_read_write 2
```

time	dev	event0	pic0	event1	pic1
2	dram0	bank_busy_stalls	193307539	mem_read_write	71400764
4	dram0	bank_busy_stalls	192952912	mem_read_write	71338904
6	dram0	bank_busy_stalls	194118606	mem_read_write	71866659
8	dram0	bank_busy_stalls	194180462	mem_read_write	71822108
10	dram0	bank_busy_stalls	129238477	mem_read_write	49908669
12	dram0	bank_busy_stalls	1029510	mem_read_write	641610
14	dram0	bank_busy_stalls	10229	mem_read_write	11562

modify consistent_coloring setting

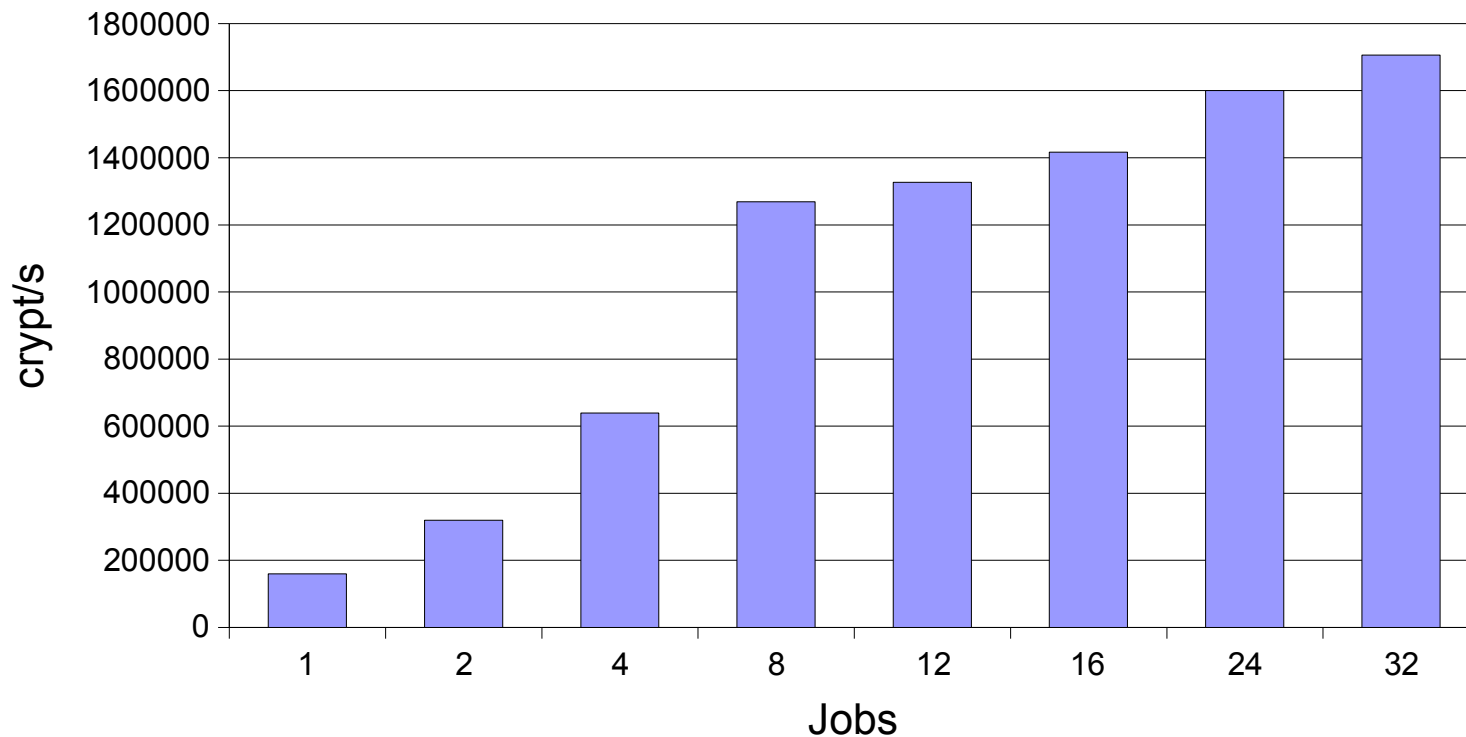
2	dram0	bank_busy_stalls	524127	mem_read_write	435160
4	dram0	bank_busy_stalls	515648	mem_read_write	430071
6	dram0	bank_busy_stalls	946224	mem_read_write	692673
8	dram0	bank_busy_stalls	970769	mem_read_write	711966
10	dram0	bank_busy_stalls	13895124	mem_read_write	7812990
12	dram0	bank_busy_stalls	846321	mem_read_write	528989
14	dram0	bank_busy_stalls	3968	mem_read_write	5553

Looking Into The Black-Box- how the kernel may impact your application



This Scales!

- the scaling of 10.2 for 32 jobs is much better now



Scaling Problem Solved

- know your tools
- know your hardware
- deploy kernel patches
 - scaling problem solved by patch in the meantime;
changes the default allocation behavior



Case Study #2

Memory Placement

Optimizations

Memory Placement Optimization

- keyword MPO
- Solaris is smart about allocating memory on NUMA architectures (e.g. Opteron SMP)
 - memory is allocated local to the thread executing the code (first touch)
- keep in mind that *malloc(3c)* does not return memory to the operating system
- DTrace *sched* provider reveals information about locality groups

Looking Into The Black-Box- how the kernel may impact your application

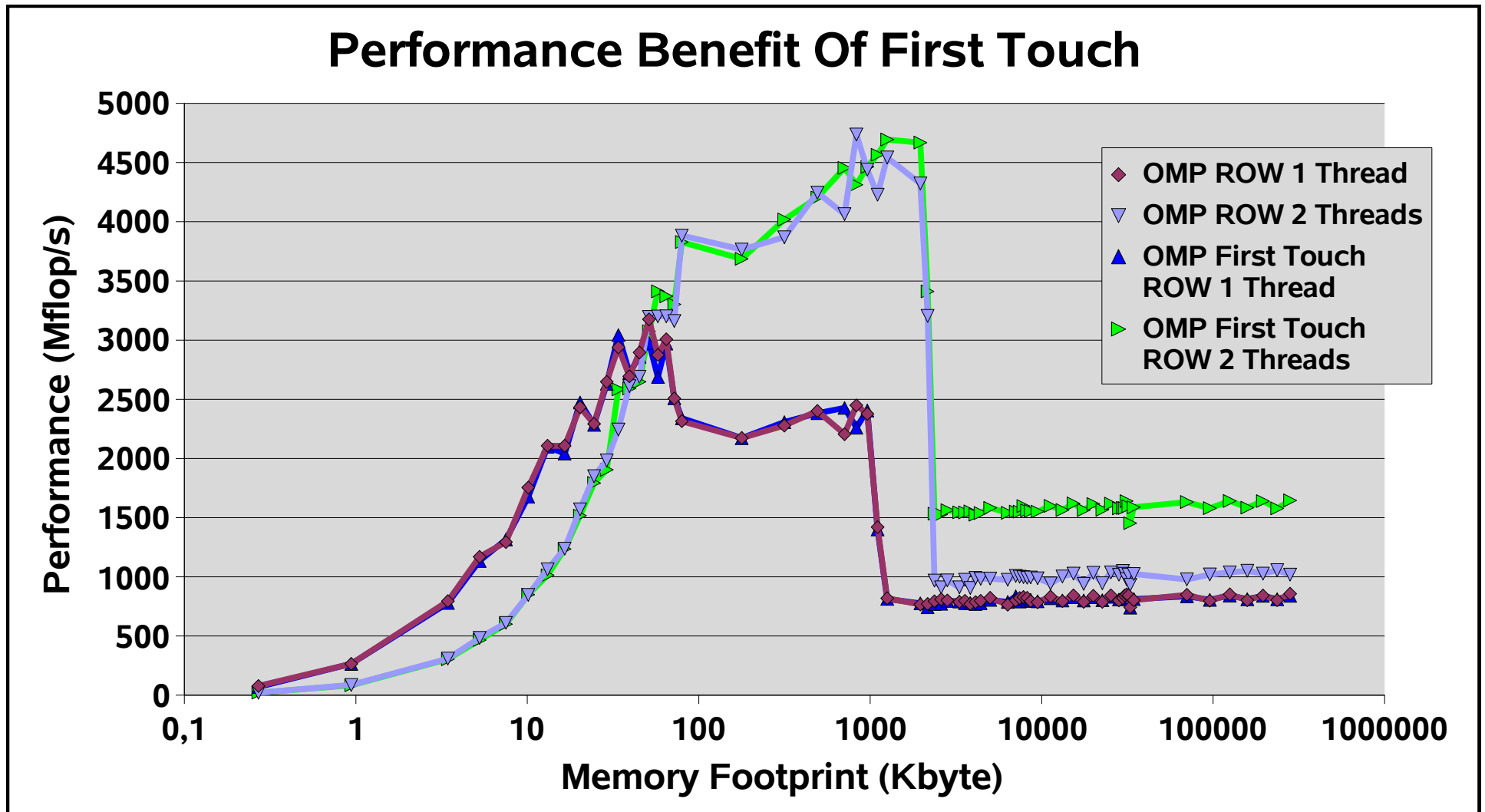


chart courtesy of Rud van der Pas

Looking Into The Black-Box- how the kernel may impact your application



**Thank you, and
remember ...**

*Use The
DTrace!*

