OpenMP®

SC25 OpenMP Tech Talk Series

Using DTrace to analyze your OpenMP application

Ruud van der Pas
Senior Principal Software Engineer, Oracle Linux Engineering

Outline

- Motivation
- What is DTrace?
- DTrace Mini Tutorial
- Analyzing an OpenMP Application with DTrace
- Wrap Up

Motivation

Motivation

- **The DTrace tool has been around for quite some time**
  - Focus so far has been on analyzing what the OS does
- **There is a lot of potential to also use it for applications though**
  - Can analyze from the application deep into libraries and the OS
  - Full and transparent support for multithreaded applications
- **Started to use this on OpenMP applications**
  - Although early work, the first results are encouraging

What is DTrace?

DTrace - What is it?

- **DTrace is a dynamic tracing tool**
- **Traces activities in userland, the OS kernel, or both**
  - Can combine both for top-to-bottom "holistic" tracing
- **Full control over what is traced, and when**
  - Typically zoom in on an area of interest
- **Designed to be safe to use on a production system**
- **Easy to use**

DTrace - Main Features

OpenMP

- **Programmable tracing using scripts written in the D language**
  - The D language has elements of awk, OS shells, and C
- **Provides flexibility, plus access to detailed information**
  - For example, print the arguments of a function call
- **Flexible reporting and lay-out**
  - The user controls what to print and when

DTrace Mini Tutorial

DTrace - Key Concepts and Terminology

- A DTrace probe is an event that can be traced
  - Every system has a set of probes you can chose from
  - For example, trace all calls to the malloc() function
- Probes are organized in sets, called providers
- For example:

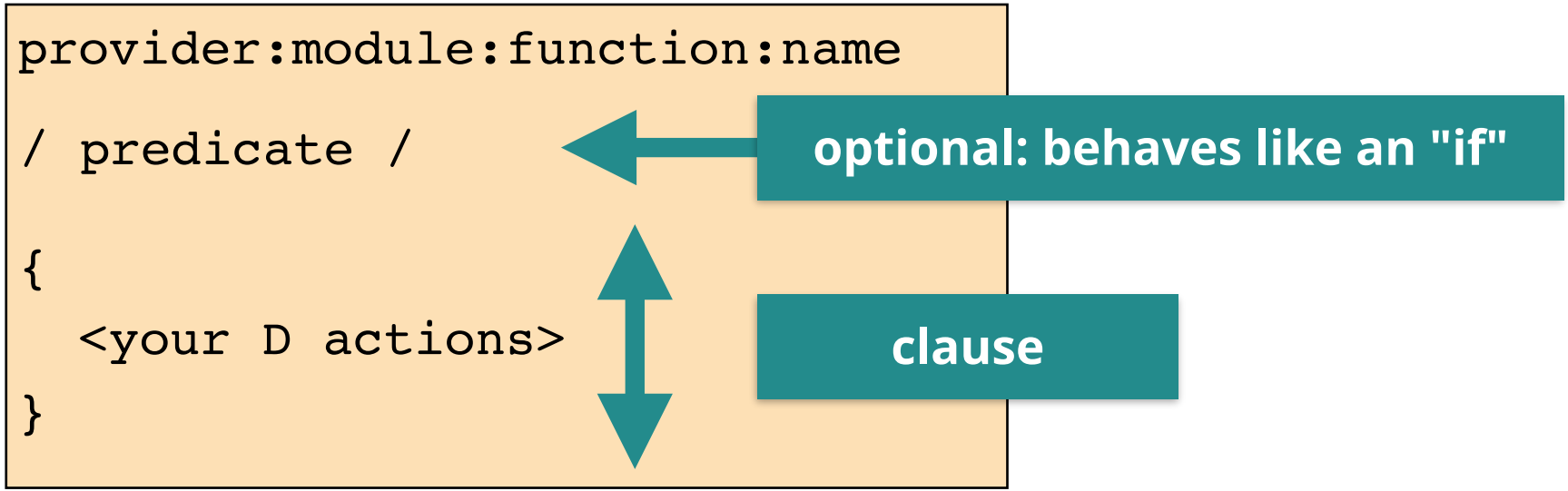
```
$ sudo dtrace -l

   ID   PROVIDER     MODULE   FUNCTION      NAME
                                  ...
 4020    syscall   vmlinux      getcwd     entry
 4019    syscall   vmlinux      getcwd    return
                                  ...
$
```

DTrace Probes and Clauses

OpenMP

- **The user specifies the probe(s) to be monitored**
  - If the probe conditions are met, the probe is said to "fire"
- **The D language is used to program the action(s) to be taken**
- **Actions are grouped in a clause**
  - A clause consists of one or more D statements
  - A single probe may have more than one clause
  - Probes may share a clause

The Definition of a DTrace Probe



- A DTrace script, or program, consists of a collection of probes
- Each probe definition is associated with one or more clauses
- The optional predicate controls execution of the clause

Examples of a DTrace Probe

OpenMP

- The probe definition has 4 fields:

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

```
syscall:vmlinux:getcwd:entry
```

- Fields may be omitted and act as a wildcard
- Wildcards within a field are also supported
- An example omitting a field and using a wildcard:

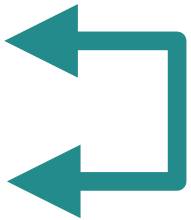
```
syscall::write*:return
```

DTrace Predicates

- Predicates are used to control the flow in a D script
- A predicate is a logical expression
  - It is enclosed in forward slashes: / <logical expression> /
  - The expression evaluates to TRUE/non-zero, or FALSE/zero
  - If TRUE: execute the associated clause
- An example:

```
/ var_has_been_set /
```

```
/ var_has_been_set == 1 /
```



equivalent

DTrace Aggregations

OpenMP

- **Reduces many data values into a compact aggregate**
  - For example, the minimum of a set of numbers
  - An aggregation stores the result of an aggregation function
- **Aggregations in DTrace start with the @ symbol:**
  - Example:
```
@my_first_aggregation = min(my_var);
```
- **Aggregations may be indexed through a (multi-valued) key:**
  - Example:
```
@my_second_aggregation["a",123] = min(my_var);
```

How to Run Your DTrace Program

OpenMP

- **There are two ways to run your DTrace script:**
  - Execute the script from the command line
  - Store the script in a file and execute it
- **We focus on the second method**
  - The convention is to use .d as a file extension (e.g. myscript.d)
- **Need to run a DTrace script as root/sudo:**

```
$ sudo dtrace -s myscript.d
```

Analyzing an OpenMP Application with DTrace

The Experiment

- **DTrace was used to analyze a graph analysis application**
  - Written in C, parallelized with OpenMP, compiled with gcc
  - This means that the GOMP library called libgomp.so is used
  - This library is the module specified in our probes
- **Operational:**
  - To keep the output manageable, used 2 OpenMP threads
  - Run time of test: 7.02user 0.30system 0:03.82elapsed 191%CPU
- **Wrote two DTrace scripts for the analysis**
  - As usual with DTrace, cast a wide net and then zoom in

The Two DTrace Scripts Used

OpenMP

- **File name: sc25-scan.d**
  - Casts a wide net to see which GOMP functions are called
  - Also include the OpenMP run time functions starting with "omp_"
  - Leverage the wildcard feature
- **File name: sc25-omp.d**
  - Zooms in on some functions found in the first step
  - Display several thread statistics and timings
- **Run command for both:**

```
$ sudo dtrace -s <script> -c ./graph-code
```

sc25-scan.d - Setup and Key Probes

OpenMP

```
 1 #pragma D option quiet
 2 #pragma D option aggsortkey=1
 3 #pragma D option aggsortkeypos=0
 4
 5 pid$target:libgomp.so:GOMP*:entry,
 6 pid$target:libgomp.so:omp_*:entry
 7 {
 8   @calls_total[probefunc]       = count();
 9   @calls_per_thr[tid,probefunc] = count();
10 }
```

customize some settings

store and count how often the target functions are called

two aggregations

count() is incremented each time the probe fires

sc25-scan.d - Print the Results

OpenMP

```
11 END
12 {
13   printf("\n%-20s  %7s\n\n","Function","Count");
14   printa("%-20s  %7@d\n",@calls_total);
15   printf("\n%10s  %-20s  %6s\n\n","TID","Function","Count");
16   printa("%10d  %-20s  %6@d\n",@calls_per_thr);
17 }
```

- The END probe fires once, after the tracing has completed
- Use printf() to print headers for the tables
- The printa() function is used to print the aggregations*

*) Unless explicitly printed, aggregations are always printed by default

The Results of the Scan

OpenMP

Total Function Call Counts

Function	Count
GOMP_barrier	3896
GOMP_parallel	197
GOMP_single_start	814
omp_get_num_threads	396
omp_get_thread_num	396

Call Counts Differentiated by Thread

TID	Function	Count
367832	GOMP_barrier	1948
367832	GOMP_parallel	197
367832	GOMP_single_start	407
367832	omp_get_num_threads	198
367832	omp_get_thread_num	198
367833	GOMP_barrier	1948
367833	GOMP_single_start	407
367833	omp_get_num_threads	198
367833	omp_get_thread_num	198

sc25-omp.d - Setup and First Probe

OpenMP

```
 1 #pragma D option quiet
 2 #pragma D option aggsortkey=1
 3 #pragma D option aggsortkeypos=0
 4
 5 pid$target:libgomp.so:omp_get_thread_num:return
 6 / thr_id_set[tid] == 0 /
 7 {
 8   this->omp_tid                    = arg1;
 9   omp_thr_id[tid]                  = this->omp_tid;
10   @thr_id_map[this->omp_tid,tid] = count();
11   thr_id_set[tid]                  = 1;
12 }
```

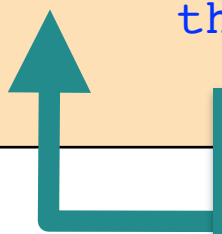
record and store both thread IDs

- Get the DTrace thread ID stored in variable tid
- Line 8: capture the OMP TID as the return value of omp_get_thread_num()

Block #2 - Populate @thread_stats

OpenMP

```
13 pid$target:libgomp.so:omp_get_num_threads:return
14 {
15   this->thr_map_set = (thr_id_set[tid] == 1) ?
16                                       omp_thr_id[tid] : -1;
17   this->thr_count   = arg1;
18   @thread_stats[probefunc,this->thr_count,tid,
19                 this->thr_map_set] = count();
20 }
```



use 4 fields as a key:

the function name, the number of threads returned, and the two thread IDs

Block #3 - Count Calls and Start Timer

```
21 pid$target:libgomp.so:GOMP_parallel:entry,
22 pid$target:libgomp.so:GOMP_barrier:entry,
23 pid$target:libgomp.so:GOMP_single_start:entry
24 {
25   @call_counts[probefunc,tid] = count();
26   self->ts_start[probefunc]   = timestamp;
27 }
```

- When one of these functions is called, the probe fires
- Count and store in @call_counts how often this happens
- The reference time is recorded in self->ts_start
- This is a thread-local associative array

Block #4 - Get Timing Statistics/1

OpenMP

```
28 pid$target:libgomp.so:GOMP_parallel:return,
29 pid$target:libgomp.so:GOMP_barrier:return,
30 pid$target:libgomp.so:GOMP_single_start:return
31 / self->ts_start[probefunc] /
32 {
33   this->time = (timestamp - self->ts_start[probefunc])/1000;
```

- These probes fire on return of the functions
- The predicate at line 31 ensures that the timer for the function has been set
- At line 33, we compute the time passed since the entry probe fired
- This time is in nanoseconds, convert to microseconds

Block #4 - Get Timing Statistics/2

OpenMP

```
35    @t_total[probefunc,tid]      = sum(this->time);
36    @t_histogram[probefunc,tid] = quantize(this->time);
37    @t_min[probefunc,tid]        = min(this->time);
38    @t_max[probefunc,tid]        = max(this->time);
39    @t_stddev[probefunc,tid]     = stddev(this->time);
40
41    self->ts_start[probefunc] = 0;
42  }
```

- At lines 35-39 we compute and store the timing statistics
- The key consists of the function name and DTrace thread ID
- At line 41, the persistent storage for the thread-local variable is released

END Probe - Print the Results/1

OpenMP

```
43 END
44 {
45 printf("-------------------------------\n");
46 printf("%12s %12s %5s\n","OpenMP TID","DTrace TID","Count");
47 printf("-------------------------------\n");
48 printa("%7d %15d %7@d\n",@thr_id_map);
49
50 printf("\n-------------------------------");
51 printf("-------------------------------\n");
52 printf("%20s %12s %12s %9s %6s\n","Function name",
53                 "Thr. Count","DTrace TID","OMP TID","Count");
54 printf("-------------------------------");
55 printf("-------------------------------\n");
56 printa("%-20s %7d %17d %6d %9@d\n",@thread_stats);
57 printf("\nNOTE: OMP TID = -1 => mapping was not set (yet)
```

END Probe - Print the Results/2

OpenMP

```
59 printf("%38s %s\n"," ","|--------- Time in us ---------|");
60 printf("%20s %8s %8s %8s %6s %7s %7s\n\n","Function name",
61                                 "TID","Count","Total","Min","Max","Stddev");
62 printa("%-20s %8d %8@d %8@d %6@d %7@d %7@d\n",@call_counts,@t_total,
63                                                 @t_min,@t_max,@t_stddev);
64
65 printf("\n");
66 printa("Time in useconds in %s - TID = %d %@d\n",@t_histogram);
67 }
```

The Results - Thread Statistics

```
-------------------------------
  OpenMP TID   DTrace TID Count
-------------------------------
       0          367862     1
       1          367863     1

-----------------------------------------------------------------------
       Function name   Thr. Count   DTrace TID   OMP TID  Count
-----------------------------------------------------------------------
omp_get_num_threads          2          367862       0        197
omp_get_num_threads          2          367862      -1          1
omp_get_num_threads          2          367863       1        197
omp_get_num_threads          2          367863      -1          1

NOTE: OMP TID = -1 => mapping was not set (yet)
```

The Results - Timing Summary

OpenMP

Function name	TID	Count	Time in us Total	Min	Max	Stddev
GOMP_barrier	367862	1948	34058	4	12283	284
GOMP_barrier	367863	1948	605968	4	11475	1509
GOMP_parallel	367862	197	2837841	877	620127	47696
GOMP_single_start	367862	407	2075	4	19	1
GOMP_single_start	367863	407	2006	4	17	3

- The times in the parallel region and barriers have quite a variation
- The times in the two barriers are not balanced
- On average: 10 barriers calls per thread for one parallel region invocation

The Results - Timing Histograms/1

OpenMP

```
Time in useconds in GOMP_parallel - TID = 367862
           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@@@@@@@@@@@                           64
            1024 |                                         1
            2048 |                                         0
            4096 |                                         1
            8192 |@@@@@@@@@@@@@@@                          67
           16384 |@@@@@@@@@@@@@                            61
           32768 |                                         1
           65536 |                                         0
          131072 |                                         0
          262144 |                                         1
          524288 |                                         1
         1048576 |                                         0
```

The Results - Timing Histograms/2

```
Time in useconds in GOMP_barrier - TID = 367862
value  ------------- Distribution ------------- count
    2 |                                         0
    4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  1671
    8 |@@@@                                     202
   16 |@                                        32
   32 |                                         6
   64 |                                         14
  128 |                                         13
  256 |                                         4
  512 |                                         2
 1024 |                                         3
 2048 |                                         0
 4096 |                                         0
 8192 |                                         1
16384 |                                         0
```

```
Time in useconds in GOMP_barrier - TID = 367863
value  ------------- Distribution ------------- count
    2 |                                         0
    4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  1654
    8 |@@                                       81
   16 |                                         19
   32 |                                         24
   64 |@                                        34
  128 |                                         21
  256 |                                         12
  512 |                                         14
 1024 |                                         15
 2048 |                                         12
 4096 |@                                        25
 8192 |@                                        37
16384 |                                         0
```

- Many barrier calls are fast, but there are significant outliers
- The data points at a load balancing issue in this application

The Results - Timing Histograms/3

```
Time in useconds in GOMP_single_start - TID = 367862
           value  ------------- Distribution ------------- count
 2 |                                         0
 4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  397
 8 |@                                        8
16 |                                         2
32 |                                         0

Time in useconds in GOMP_single_start - TID = 367863
           value  ------------- Distribution ------------- count
 2 |                                         0
 4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  401
 8 |                                         5
16 |                                         1
32 |                                         0
```

- The time spent in the single region is short
- There are no significant outliers

Summary

- **With our two DTrace scripts we found that:**
  - Each thread calls omp_get_num_threads() 198 times, but no change
  - On average a high barrier call count per parallel region invocation
  - The time spent in the parallel region has some long outliers
  - The time spent in the barrier is not balanced across the threads
- **This points at a potential for improvement**
  - Dig deeper with DTrace
  - Use an application profiling tool to analyze at the source code level

Wrap Up

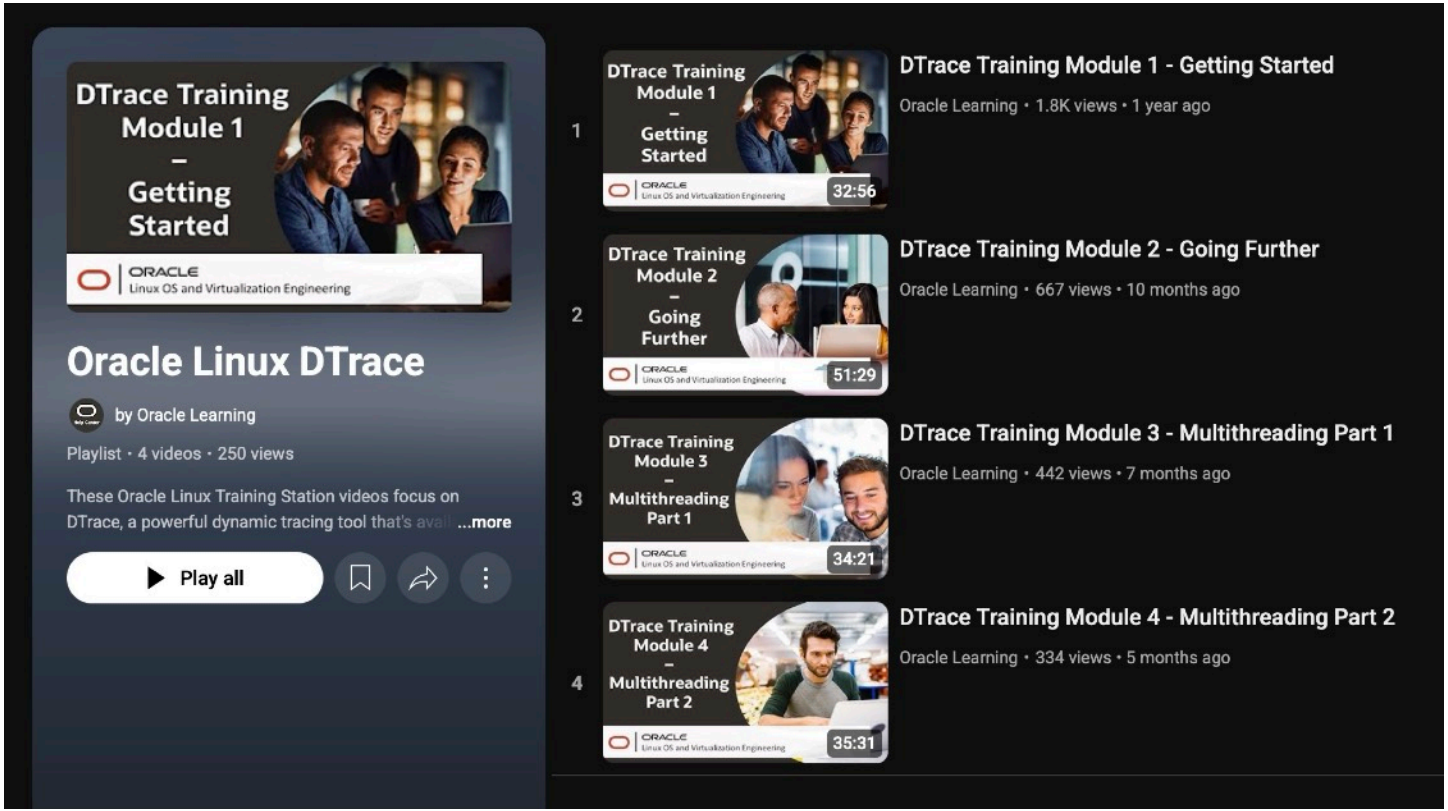
Conclusions

- **DTrace provides quite some insight**
  - No need to change the executable
  - Can analyze production runs
- **DTrace is very useful to obtain detailed run time statistics**
  - A great way to get deep insight into the OpenMP behavior
- **The scripts shown here are just the start**
  - Much more to be uncovered!

Acknowledgements

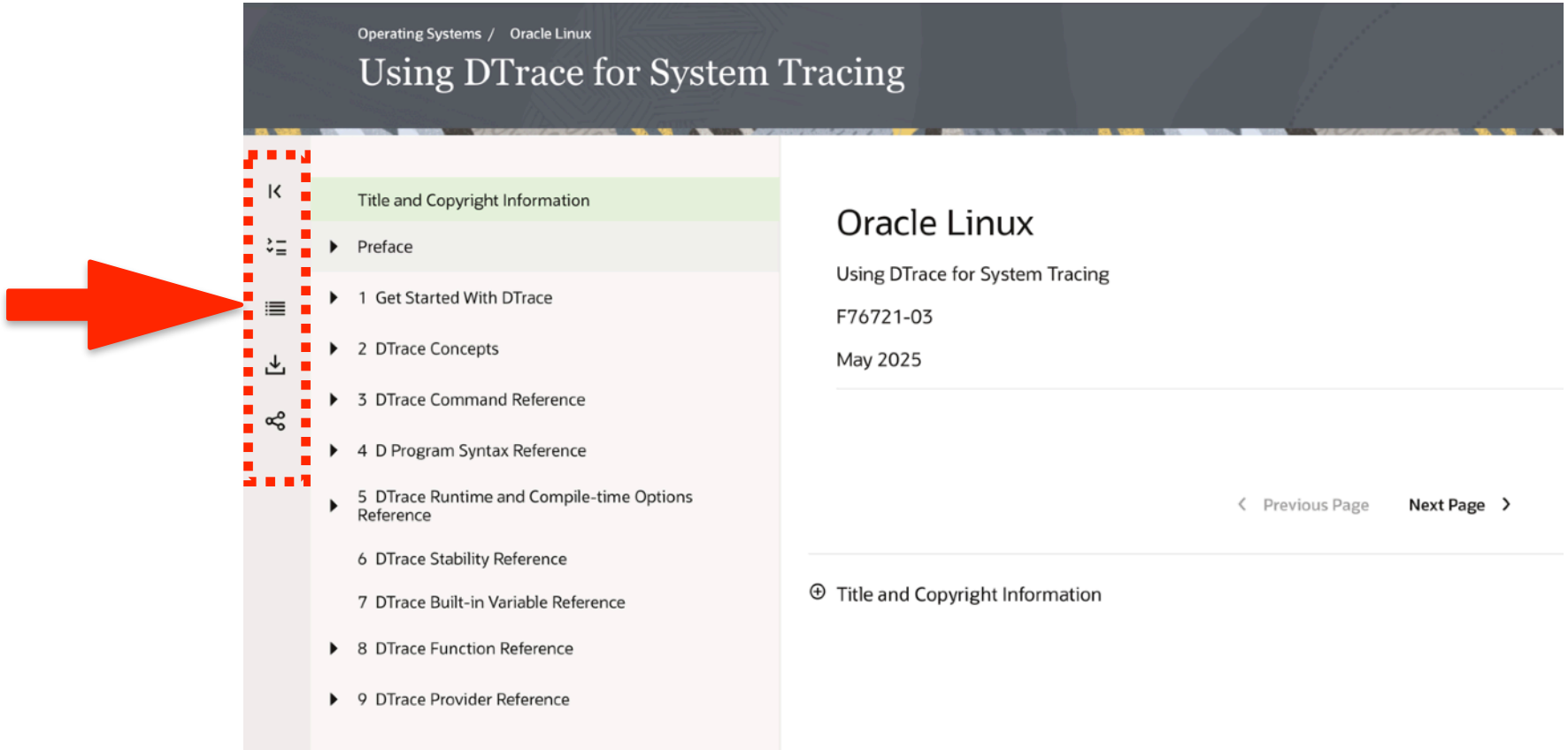
- **I would like to thank the following persons**
  - Jakub Jelinek at RedHat for his help with my libgomp questions
  - Eugene Loh, Kris van Hees, Nick Alcock, and Alan Maguire at Oracle for our very fruitful discussions on DTrace
- **Without them, this presentation could not have been made**
  - And any errors are entirely mine

DTrace Trainings on YouTube



`https://www.youtube.com/@OracleLearning/playlists -> Oracle Linux DTrace`

The DTrace User Guide



https://docs.oracle.com/en/operating-systems/oracle-linux/dtrace-v2-guide

Thank You And … Stay Tuned!

Bad OpenMP Does Not Scale

Ruud van der Pas

SC25 OpenMP Tech Talk

openmp.org OpenMP API specs, forum, reference guides, and more

link.openmp.org/sc25talks SC25 OpenMP Tech Talk videos and presentations