DTrace Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Document serial number: 000004892117

Table of Contents

1 DTrace Overview 3
Characteristics 3
Files 3
Sample Programs 3
2 Using DTrace 4
DTrace Providers and Probes 5
Environment Variables 6
3 DTrace Operation 8
4 Coordination between DTrace and the Host Tool 10
breakpoint() 10
Coordination with the Debugger 10
Coordination with Razor 11
5 Examples 12
Example 1: Probe List 12
Example 2: Hello World 12
Example 3: Profile Provider 12
Example 4: System Call Count 12
Example 5: Total System Call Time 12
Example 6: System Call Time per Frame 13
Example 7: Monitoring the IO/File Manager 13

1 DTrace Overview

Characteristics

SCE DTrace is a comprehensive dynamic tracking facility that you can use to examine and understand the behavior of the operating system and of applications on a target Development Kit. The DTrace tool is based on open-source DTrace, developed by Sun Microsystems, Inc.

You use DTrace to see how your application works, track down performance problems, and locate causes of aberrant behavior. The current implementation of DTrace provides a large number of static probes in the kernel and drivers. You can also add static user-space probes to your code. These probes only have a measurable performance impact when enabled. Dynamic user-space probes will be made available at a later date.

With dynamic probes, instrumentation occurs dynamically at runtime; that is, DTrace will dynamically patch instructions to create probes. Keep in mind that there is some performance overhead when probes are enabled, regardless of whether the probes are dynamic or static. However, when dynamic probes are disabled there is no overhead at all since the code is not modified.

Files

The DTrace command line tool is called dtrace.exe, and it is part of the SDK. The dtrace.exe command line tool is installed by the SDK into the following location:

```
%SCE_PSP2_SDK_DIR%\host_tools\bin\
```

The DTrace CLI (command line interface) is an optional install as part of the normal installation of the SDK. The SDK Manager copies the DTrace executable and all necessary libraries to the correct locations on a Windows system. To run DTrace scripts and programs, you need to open a Windows terminal (also called a DOS shell) and type in:

```
%SCE_PSP2_SDK_DIR%\host_tools\bin\dtrace.exe
```

To use DTrace, you can either supply scripts directly on the command-line or by writing D language script files.

Sample Programs

Chapter 5, “Examples” contains sample scripts and programs that illustrate using dtrace.exe.

2 Using DTrace

You use DTrace to activate instrumentation points, called probes, via the DTrace command-line tool. Probes are published by providers, which typically correspond to a kernel or user-space module. For example, the `profile` provider offers probes associated with time-based interrupts that fire at fixed, specified time intervals. You use these probes to sample various aspects of system state every interval, from which you can infer system behavior.

You can invoke DTrace by supplying a request directly on the command line (`-n` option) or via a script file (`-s` option). DTrace accepts scripts written in the D language, which is a language structured similarly to C. The D language is derived from a subset of C, with the addition of special tracing functions and types.

Each DTrace probe is designed to be activated by a specific behavior. This is called firing the probe. Firing a probe lets you retrieve relevant information about the particular behavior and also enables you to specify an action for DTrace to take. You write predicates and actions, which you program with the DTrace D language, to record information of interest to you when probes fire in the operating system kernel and user processes. A probe is a location or activity to which DTrace binds a request to perform a set of actions, like recording a stack trace, a timestamp, or the argument to a function. When a probe fires, DTrace gathers the data from the probe and reports the data to you. In addition to reporting specific actions, a probe can be set to fire but take no action. DTrace keeps a log of each time a probe fires.

A probe is identified by its probe description, a unique four-tuple identifier: `provider:module:function:name`. The `provider` component indicates the name of the provider that makes the probe available. The `module` and `function` components identify the module and function, respectively, that the probe instruments. The `name` component gives the actual name of the probe.

You can fully specify a probe by naming each four-tuple component that uniquely identifies that probe. The only required element of a probe specification is the `name` component. Leaving a component entry empty in the probe specification acts like a wildcard: that is, that component matches anything. For example, the following probe specification leaves the `function` component empty: `syscall:SceKernelThreadMgr::entry`. This specification indicates that the probe must be from the `syscall` provider. In addition, the probe must be in the `SceKernelThreadMgr` module. (Had the module component been left empty, then the probe could be from any module.) The `function` component, because it is empty, is a wildcard, matching any probe function name in the `SceKernelThreadMgr` module. Finally, the `entry name` component indicates that the probe must be named `entry`.

To use a probe, you need to build a DTrace request or program. You can construct a request on the command-line and execute it immediately, or place it in a text file as a D program and then invoke DTrace with that text file. Furthermore, you can use any number of probes and actions in a DTrace program or request. There are examples of both DTrace requests and programs in Chapter 5, “Examples”.

DTrace programs, whether specified on the command line (with the `-n` option) or through scripts (with the `-s` option), may also include action statements and predicates. Scripts are typically used for more complex requests, as such complex requests can be unwieldy to specify on the command line.

Action statements are user-programmable statements that DTrace executes within the kernel. Actions are taken when a probe fires. In addition, actions are completely programmable using D. You may include multiple action statements, separated by semi-colons, in the probe action section.

The predicate clause, also optional, contains expressions and is enclosed in slashes. Predicates appear in a D program before the action clauses and after the probe description. Predicates are the basic conditional construct for building complex control flow in a D program. These expressions refer to D data objects, such as variables and constants, and can use any valid D operator. At the time that a probe fires, DTrace evaluates the predicate expressions to determine whether to execute associated actions. You may omit the predicate section entirely for any probe. When the predicate is omitted, firing a probe always results in

executing the probe actions. If you do use a predicate, the predicate expression must evaluate to an integer value or a pointer type. Also, as with all D expressions, a zero value indicates false and any non-zero value evaluates to true.

DTrace Providers and Probes

The following providers and probes are included in the SCE version of DTrace.

Available Providers

The providers summarized in Table 1 are currently supported by SCE DTrace.

Table 1 Available SCE DTrace Providers

Provider	Description
`dtrace`	Provides probes related to DTrace, such as for initializing state before tracing begins.
`proc`	Provides probes pertaining to processes.
`profile`	Provides probes pertaining to time-based interrupt firing.
`sched`	Provides probes related to CPU scheduling.
`syscall`	Provides probes at the entry to and return from every system call.
`interrupt`	Provides probes associated with hardware interrupt events.
`usdt` (User-Level Statically Defined Tracing)	Provides the ability to define custom static probes in application code.

These providers make probes available that you can use for the following types of tasks:

- dtrace provider – The dtrace provider provides probes related to DTrace itself. Use these probes to initialize state before tracing begins, to process state after tracing has completed, and to handle unexpected execution errors in other probes.
- proc provider – The proc provider makes available probes pertaining to the following activities: creating and terminating processes, executing new program images, and thread creation and termination. Note that thread-related probes have the prefix lwp (light weight process) in order to retain the same probe names as on other DTrace platforms.
- profile provider – The profile provider provides probes associated with time-based interrupts firing at fixed, specified time intervals. These unanchored probes are not associated with any particular point of execution, but rather they are associated with the asynchronous interrupt event. You can use these probes to sample some aspect of system state every unit time. Using the collected samples you can infer system behavior. Generally, if you specify a high sampling or a long sampling time, an accurate inference is possible. By using the profile provider in conjunction with DTrace actions, you can sample many things in the system. For example, you could sample the state of the current thread, the state of the CPU, or the current machine instruction. However, some things (such as many types of kernel state and system process state) are off-limits for inspection.
- sched provider – The sched provider makes available probes related to CPU scheduling. Because CPUs are the one resource that all threads must consume, the sched provider is very useful for understanding systemic behavior. For example, using this provider, you can understand when and why threads sleep, run, change priority, or wake other threads.
- syscall provider – The syscall provider makes available two probes, one at the entry to and another at the return from every system call in the system. Because system calls are the primary interface between user-level applications and the operating system kernel, you can use this provider to gain tremendous insight into application behavior with respect to the system.

- interrupt provider – The interrupt provider makes available probes associated with an interrupt handler execution. A probe fires when an event of a given type has occurred. Except for the vblank probe, the interrupt provider has a pair of probes for each interrupt handler function: An entry probe that fires when the interrupt handler is entered, and a return probe that fires after the handler has completed but before control transfers back to the previous context. For all interrupt probes except vblank, the function name is the name of the instrumented interrupt handler, plus, the module name is not defined.
- usdt (User-Level Statically Defined Tracing) provider – Although the usdt provider does not include any probes, it allows you to define your own static probes in the application code.

Available Probes

The probes listed in Table 2 are currently available in DTrace. Table 2 lists these probes according to the respective providers.

Table 2 Available SCE DTrace Probes

Provider	Probe
`dtrace`	`BEGIN`, `END`, `ERROR`
`proc`	`create`, `exec`, `exec-failure`, `exec-success`, `exit`, `lwp-create`, `lwp-start`, `lwp-exit`, `start`
`profile`	`profile-n`, `tick-n`
`sched`	`change-pri`, `dequeue`, `enqueue`, `off-cpu`, `on-cpu`, `preempt`, `remain-cpu`, `sleep`, `wait`, `wakeup`
`syscall`	`entry`, `return` for each system call
`interrupt`	`vblank`; `entry`, `return` for other interrupt handlers
`usdt` (User-Level Statically Defined Tracing)	Provides the ability to define custom static probes in application code.

Environment Variables

The DTrace command line tool loads the environment variables indicated in the table below. This operation is unique to the PlayStation®Vita platform.

Environment Variable	Description
`SCE_PSP2_SDK_DIR`	Automatically set upon installing the SDK. This environment variable must be set as it is used to resolve paths in the DTrace command line tool.
`SCE_PSP2_DTRACE_SELF_DIR`	Specifies the path where the SELF file is located. The DTrace command line tool searches for symbol information in the specified path upon resolving the symbol file. (Search is limited to the first level. Note that a recursive search is not performed.) When setting multiple directories, separate directories using a semicolon.
`SCE_PSP2_DTRACE_UPRX_DIR`	Specifies the path where the URPX file is located. Like `SCE_PSP2_DTRACE_SELF_DIR`, symbol information is searched for in the specified patch.

Specify an environment variable as follows and execute script to display stack while resolving symbols in the SELF file.

```
Set
SCE_PSP2_DTRACE_SELF="c:\projectA\PSVita_Debug\;c:\projectB\PSVita_Debug"
%SCE_PSP2_SDK_DIR%\host_tools\bin\dtrace.exe -s ustack_test.d
```

To add a probe to an application, first define the provider and the probe in the .d file. The following is a code example for defining the `primes` provider.

ustack_test.d Example

```
syscall:::entry
{
  ustack();
}
```

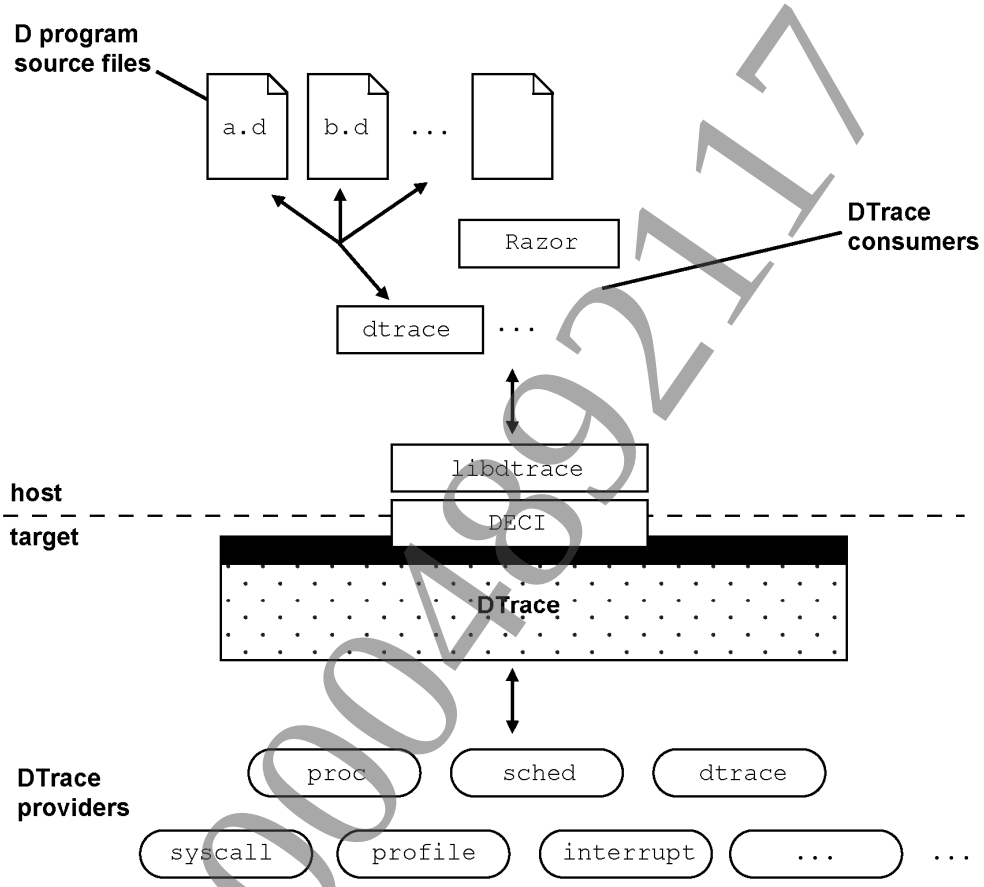
3 DTrace Operation

The DTrace tracing architecture consists of the following components:

- User-level programs; for example, dtrace. Also called consumers.
- Providers, packaged as kernel modules, that provide probes to gather tracing data.
- A library interface that user programs use to access DTrace through the dtrace kernel driver.

Figure 1 below shows a high-level view of the standard DTrace architecture.

Figure 1 DTrace Architecture



SCE DTrace is split between a host tool or library and a target kernel module. With standard DTrace, both DTrace and your application talk to the same system and pass information between the system and user-space via system calls. For SCE DTrace, run DTrace on a development host computer with Windows running and connect to the target (Development Kit) on which the application is running via USB/Ethernet. The library uses DECI packet communication to carry out communication between the development host computer and the target.

Users familiar with DTrace on a unified system might expect to see an immediate and continuous display of information on the console regardless of the script. However, because DTrace communicates with the target over USB/ethernet, you may notice a lag before information displays on the development host computer or you may notice that drops of information, rather than a continuous information feed, occur in the console feedback. Such breaks in the information display will be most evident with a script or `dtrace` command that tries to do high-frequency output. For example, consider a command or script such as:

```
dtrace.exe -n 'profile-1000 { stack(); }'
```

This script tries to print the system callstack for each processor 1000 times per second. Since acquiring the data and sending that data to the development host computer probably takes more than a millisecond, there may be drops of buffers of data. DTrace is designed to drop buffers of data to the development host computer in a situation such as this to ensure that it does not fail.

4 Coordination between DTrace and the Host Tool

breakpoint()

Coordination between Dtrace and the host tool can be realized by using a `breakpoint()` action. `breakpoint()` action can be used by the `return` probe of the `syscall` provider and by the `usdt` provider.

When the `breakpoint()` action is generated, DTrace writes a unique software break to the user program. For the `syscall` provider, the break is written where `syscall` completes; for the `usdt` provider, it is written at the next program counter after `usdt` is issued.

When a software break unique to DTrace occurs, DTrace checks whether the debugger or Razor for PlayStation®Vita (Razor) is attached to the target. If attached, an exception unique to DTrace is notified to the debugger/Razor.

In other words, the `breakpoint()` action can be used for the predication of DTrace to notify an exception to the debugger/Razor under specific conditions to stop it.

Coordination with the Debugger

Coordination with the debugger of Visual Studio is exemplified below.

The example uses the `usdt` sample described in the "DTrace Reference" document, with the `breakpoint()` action occurring in the `usdt` probe.

Select **Debug(D) –> Start Debugging(S)**, attach the debugger and run the program.

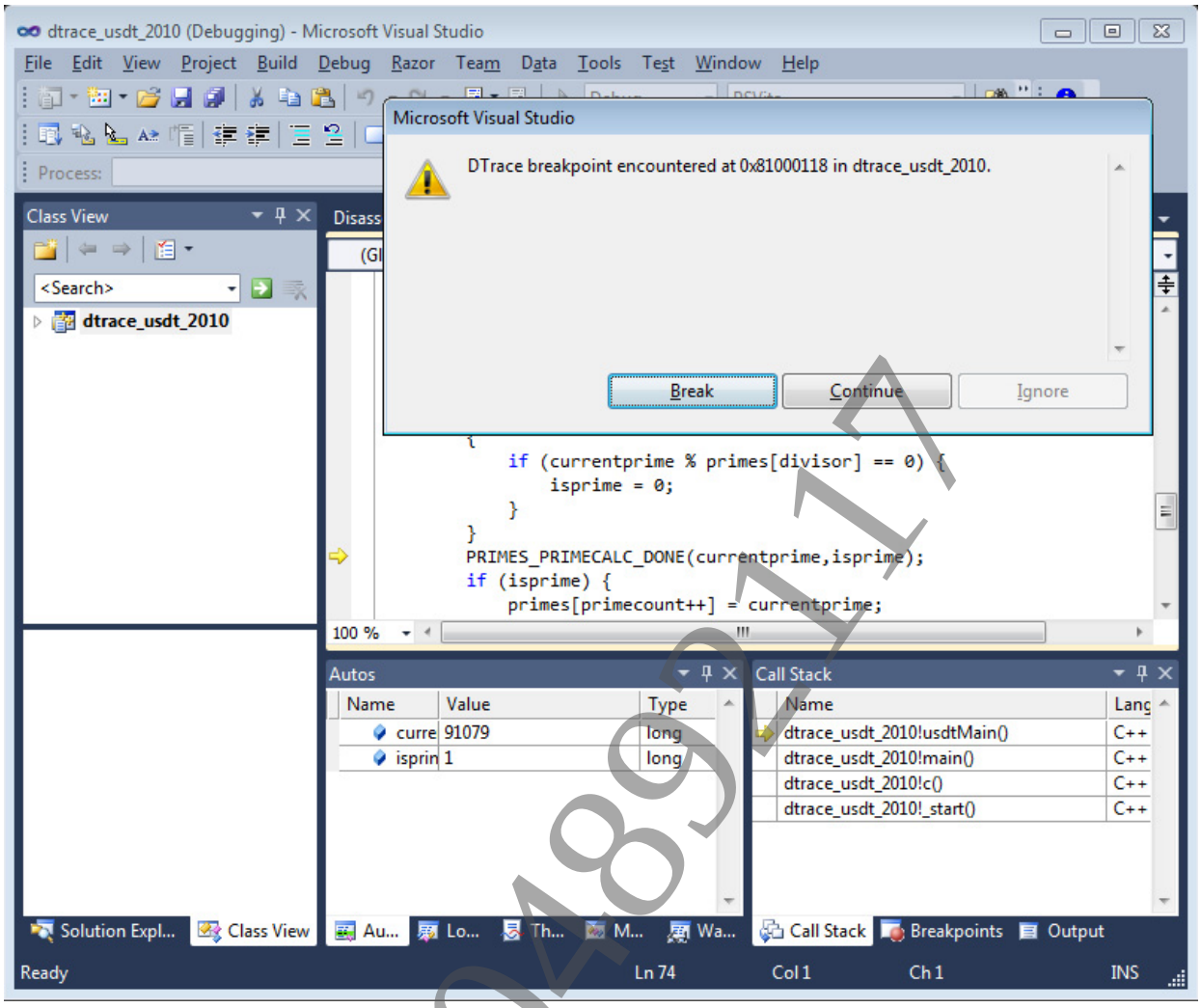
Next, execute the `dtrace` command as follows at the command prompt.

```
%SCE_PSP2_SDK_DIR%\host_tools\bin\dtrace.exe -w -n "primes*:::primecalc-done
{ breakpoint(); }"
```

When executed, the program will stop at the `primecalc-done` probe, `PRIMES_PRIMECALC_DONE()`, as shown in Figure 2.

Select **Break** to check the value of the automatic variable (for example,) in the same manner as a normal breakpoint of the debugger. Moreover, execute **Debug(D) -> Continue(C)** to continue the program. To stop tracing by DTrace, enter Ctrl-C at the command prompt running DTrace.

Figure 2 Coordination between DTrace and the Debugger



Coordination with Razor

Coordination with Razor can be realized in the same manner as the debugger.

First, execute **Razor(R) -> Capture(C) -> CPU Trace** to run CPU trace on Razor. Execute the `dtrace` command at the command prompt, as follows.

```
%SCE_PSP2_SDK_DIR%\host_tools\bin\dtrace.exe -w -n "primes*:::primecalc-done
{ breakpoint(); }"
```

When an exception unique to DTrace is notified to Razor, Razor automatically stops CPU trace. However, note that a certain amount of time will be required between the execution of the `breakpoint()` action and the stopping of trace.

5 Examples

This chapter lists some typical DTrace example scripts, with explanations. Each example shows the dtrace.exe command with option(s), plus probes and action and predicate clauses, where appropriate. The command is followed by short descriptions of example usage and expected output.

Example 1: Probe List

```
dtrace.exe -l
```

This command verifies communication between the host and targets and lists the probes that are available on the target. The output may take several seconds to complete because there may be thousands of probes. You can limit the list of probes to a particular provider with the '`-P`' option:

```
dtrace.exe -P sched -l
```

This lists only the probes in the '`sched`' provider.

Example 2: Hello World

```
dtrace.exe -n "BEGIN { printf(""hello dtrace world!\n""); exit(0); }"
```

This command verifies that the target can process a D-script.

Note: the typical UNIX shell command-line tick and quote marks are altered here to work in a Windows DOS shell.

Example 3: Profile Provider

```
dtrace.exe -n "profile-1 { trace(execname); }"
```

This command prints the name of the running process on each CPU of the target once per second. The user should press Ctrl-C to quit the process.

Example 4: System Call Count

```
dtrace.exe -n "syscall:::entry { @syscallCount[probefunc] = count(); }"
```

This command sums the call count of all system call types while the script is executing.

Example 5: Total System Call Time

syscallTime.d

```
syscall:::entry
{
    self->start = timestamp;
}

syscall:::return
/ self->start != 0 /
{
    this->elapsed = timestamp - self->start;
    @syscallTime[probefunc] = sum(this->elapsed);
    self->start = 0;
}
```

```
dtrace.exe -s syscallTime.d
```

This script shows an aggregation of the time taken for each type of system call. The script uses the `timestamp` built-in variable to access the current time, the `self` keyword to create a thread-local variable for the elapsed time, and the `this` keyword for a clause-local variable. A predicate clause on the system call return ensures that a start time has been set before adding the elapsed time to the sum.

Example 6: System Call Time per Frame

syscallTimePerFrame.d

```
syscallTimePerFrame.d:
int gFrameCount; /* globals initialized with 0 */
inline string gGameProcess = "GameOfTheYear";

syscall:::entry
/ execname == gGameProcess /
{
    self->start = timestamp;
}

syscall:::return
/ execname == gGameProcess&&  self->start != 0 /
{
    this->elapsed = timestamp - self->start;
    @syscallTime[probefunc] = sum(this->elapsed);
    self->start = 0;
}

interrupt:::vblank
{
    gFrameCount++;
}

END
{
    printf("%s - system call time (in nanoseconds) per frame (%d frames)\n",
        gGameProcess, gFrameCount);
    normalize(@syscallTime, gFrameCount); /* divide by number of frames */
    /* printa(@syscallTime); */ /* implicit */
}


dtrace.exe -s syscallTimePerFrame.d
```

This script adds to the previous example by using the `vblank` probe specific to PlayStation®Vita to normalize the system call time per frame. The global variable `gFrameCount` is incremented with every frame and then used to divide each result in the `syscallTime` aggregation before the results are printed in the DTrace `END` probe.

Example 7: Monitoring the IO/File Manager

```
dtrace.exe -s %SCE_PSP2_SDK_DIR%\host_tools\samples\dtrace\iofilemgr.d
```

As a script sample, `iofilemgr.d` is provided under `%SCE_PSP2_SDK_DIR%\host_tools\samples\dtrace\`.

Execute DTrace in the above manner to trace IO/File manager APIs.