

Dynamic Tracing Support in the Java HotSpot™ Virtual Machine

Introduction

The Java™ Platform, Standard Edition 6 (Java SE 6), introduces Dynamic Tracing (DTrace) support within the Java HotSpot™ Virtual Machine. The Dynamic Tracing Framework- a part of the Solaris™ 10 Operating System - collects performance metrics by dynamically modifying the operating system kernel and user processes to record data at specific points of interest, known as *probes*. Probes, in turn, are made available through special kernel modules, called *providers*. The providers and probes included in the Java SE 6 release make it possible for DTrace to collect performance data for applications written in the Java programming language.

The Java SE 6 release contains two built-in DTrace providers: `hotspot` and `hotspot_jni`. All probes published by these providers are user-level statically defined tracing (USDT) probes, accessed by the PID of the Java HotSpot Virtual Machine process.

The `hotspot` provider contains probes related to the following Java HotSpot Virtual Machine subsystems:

- VM Lifecycle Probes: For VM initialization and shutdown
- Thread Lifecycle Probes: For thread start and stop events
- Classloading Probes: For class loading and unloading activity
- Garbage Collection Probes: For system-wide garbage and memory pool collection
- Method Compilation Probes: Indicating which methods are being compiled, and by which compiler
- Monitor Probes: For all wait and notification events, plus contended monitor entry and exit events
- Application Probes: For fine-grained examination of thread execution, method entry/method returns, and object allocation

All hotspot probes originate in the VM library (`libjvm.so`), and as such, are also provided from programs that embed the VM.

The `hotspot_jni` provider contains probes related to the Java™ Native Interface (JNI), located at the entry and return points of all JNI methods.

In addition, the DTrace `^jstack` action prints mixed-mode stack traces including both Java method and native function names.

Inspecting Applications with DTrace and the Java Hotspot Virtual Machine

This section presents two sample applications that demonstrate the interaction of the Java SE 6 HotSpot Virtual Machine and the Solaris 10 Dynamic Tracing Framework. The first example, `Java2Demo`, is bundled with the Java SE 6 release and will already be familiar to most developers. Because the `hotspot` provider is built into the Java SE 6 VM itself, running the application is all that is required to trigger probe activity. The

1. This feature has been supported since the Java 2 Platform Standard Edition 5.0 Update Release 1

second example is a custom debugging scenario that uses DTrace to find a troublesome line of native code in a Java™ Native Interface (JNI) application.

The following script, written in the D programming language, defines the set of probes that DTrace will listen to while the Java2Demo application is running. In this case, the only probes of interest are those related to garbage collection.

hotspot_gc.d:

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

#pragma D option quiet
self int cnt;

:::BEGIN {
    self->cnt == 0;
    printf("Ready..\n");
}

hotspot$1:::gc-begin /self->cnt == 0/ {
    self->tid = tid;
    self->cnt++;
}

hotspot$1:::* / self->cnt != 0 / {
    printf("  tid: %d, Probe: %s\n", tid, probename);
}

hotspot$1:::gc-end {
    printf("  tid: %d, D-script exited\n", tid);
    exit(0);
}
```

To run this example:

1. Start the sample application: `java -jar Java2Demo.jar`
2. Note the application's ¹PID (11201 for this example)
3. Start the D script, passing in the PID as its only argument: `hotspot_gc.d 11201`

The following output shows that DTrace prints the thread ID and probe name as each probe fires in response to garbage collection activity in the VM:

Ready..

```
tid: 4, Probe: gc-begin
tid: 4, Probe: mem-pool-gc-begin
tid: 4, Probe: mem-pool-gc-begin
tid: 4, Probe: mem-pool-gc-begin
tid: 4, Probe: mem-pool-gc-begin
tid: 4, Probe: mem-pool-gc-begin
tid: 4, Probe: mem-pool-gc-end
```

¹One way to do this is with the `jps` utility, included in the J2SE™ Development Kit 5.0 and higher.

```
tid: 4, Probe: mem-pool-gc-end
tid: 4, Probe: mem-pool-gc-end
tid: 4, Probe: mem-pool-gc-end
tid: 4, Probe: mem-pool-gc-end
tid: 4, Probe: mem-pool-gc-begin
tid: 4, Probe: mem-pool-gc-begin
tid: 4, Probe: mem-pool-gc-begin
tid: 4, Probe: mem-pool-gc-begin
tid: 4, Probe: mem-pool-gc-begin
tid: 4, Probe: mem-pool-gc-end
tid: 4, Probe: mem-pool-gc-end
tid: 4, Probe: mem-pool-gc-end
tid: 4, Probe: mem-pool-gc-end
tid: 4, Probe: mem-pool-gc-end
tid: 4, Probe: mem-pool-gc-end
tid: 4, Probe: gc-end
tid: 4, D-script exited
```

The next script shows the thread ID (tid) and probe name in all probes; class name, method name and signature in the "method-compile-begin" probe; and method name and signature in the "compiled-method-load" probe:

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

#pragma D option quiet
self int cnt;

:::BEGIN {
    self->cnt == 0;
    printf("Ready..\n");
}
hotspot$1:::method-compile-begin /self->cnt == 0/ {
    self->tid = tid;
    self->cnt++;
    printf(" tid: %d, %21s, %s.%s %s\n", tid, probename,
        copyinstr(arg2), copyinstr(arg4), copyinstr(arg6));
}
hotspot$1:::method-compile-end /self->cnt > 0/ {
    printf(" tid: %d, %21s\n", tid, probename);
}
hotspot$1:::compiled-method-load /self->cnt > 0/ {
    printf(" tid: %d, %21s, %s %s\n", tid, probename,
        copyinstr(arg2), copyinstr(arg4));
}
hotspot$1:::vm-shutdown {
    printf(" tid: %d, %21s\n", tid, probename);
    printf(" tid: %d, D-script exited\n", tid);
    exit(0);
}
hotspot$1:::* / self->cnt > 0/ {
    printf(" tid: %d, %21s, %s %s\n", tid, probename,
        copyinstr(arg2), copyinstr(arg4));
}
```

Its output shows:

```
Ready..
tid: 9, method-compile-begin, sun/java2d/SunGraphics2D.setFont (Ljava/awt/Font;)V
tid: 9, compiled-method-load, setFont (Ljava/awt/Font;)V
tid: 9, method-compile-end
tid: 9, method-compile-begin, sun/java2d/SunGraphics2D.validateCompClip
tid: 9, compiled-method-load, validateCompClip ()V
tid: 9, method-compile-end
tid: 8, method-compile-begin, javax/swing/RepaintManager.addDirtyRegion0
tid: 8, compiled-method-load, addDirtyRegion0 (Ljava/awt/Container;IIII)V
tid: 8, method-compile-end
tid: 9, method-compile-begin, java/io/BufferedInputStream.read
tid: 9, compiled-method-load, read ()I
tid: 9, method-compile-end
tid: 8, method-compile-begin, java/awt/geom/AffineTransform.translate
tid: 8, compiled-method-load, translate (DD)V
tid: 8, method-compile-end
tid: 9, method-compile-begin, sun/awt/X11/Native.getInt
tid: 9, compiled-method-load, getInt (J)I
tid: 9, method-compile-end
tid: 8, method-compile-begin, sun/java2d/SunGraphics2D.setColor
tid: 8, compiled-method-load, setColor (Ljava/awt/Color;)V
tid: 8, method-compile-end
tid: 9, method-compile-begin, sun/reflect/GeneratedMethodAccessor1.invoke
tid: 9, compiled-method-load, invoke (Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/
Object;
tid: 9, method-compile-end
tid: 9, method-compile-begin, sun/java2d/SunGraphics2D.constrain
tid: 9, compiled-method-load, constrain (IIII)V
tid: 9, method-compile-end
tid: 8, method-compile-begin, java/awt/Rectangle.setLocation
tid: 8, compiled-method-load, setLocation (II)V
tid: 8, method-compile-end
tid: 9, method-compile-begin, java/awt/Rectangle.move
tid: 9, compiled-method-load, move (II)V
tid: 9, method-compile-end
tid: 8, method-compile-begin, java/lang/Number.<init>
tid: 8, compiled-method-load, <init> ()V
tid: 8, method-compile-end
tid: 8, method-compile-begin, sun/awt/X11/XToolkit.getAWTLock
tid: 8, compiled-method-load, getAWTLock ()Ljava/lang/Object;
tid: 8, method-compile-end
tid: 17, vm-shutdown
tid: 17, D-script exited
```

The next example demonstrates a debugging session with the `hotspot_jni` provider. Consider, if you will, an application that is suspected to be calling Java™ Native Interface (JNI) functions from within a *critical region*. A JNI critical region is the space between calls to JNI methods `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical`. There are some important rules for what is allowed within that space. Chapter 4 of the JNI 5.0 Specification makes it clear that within this region, "native code should not run for an extended period of time before it calls `ReleasePrimitiveArrayCritical`". In addition, "native code must not call other JNI functions, or any system call that may cause the current thread to block and wait for another Java thread".

The following D script will inspect a JNI application for this kind of violation:

```

#!/usr/sbin/dtrace -Zs

#pragma D option quiet

self int in_critical_section;

:::BEGIN {
    printf("ready..\n");
}

hotspot_jni$1:::ReleasePrimitiveArrayCritical_entry {
    self->in_critical_section = 0;
}

hotspot_jni$1:::GetPrimitiveArrayCritical_entry {
    self->in_critical_section = 0;
}

hotspot_jni$1:::*
/ self->in_critical_section == 1 / {
    printf("JNI call %s made from JNI critical region\n", probename);
}

hotspot_jni$1:::GetPrimitiveArrayCritical_return {
    self->in_critical_section = 1;
}

syscall:::entry
/ pid == $1 && self->in_critical_section == 1 / {
    printf("system call %s made in JNI critical region\n", probefunc);
}

```

Output:

```

system call brk made in JNI critical section
system call brk made in JNI critical section
system call ioctl made in JNI critical section
system call fstat64 made in JNI critical section
JNI call FindClass_entry made from JNI critical region
JNI call FindClass_return made from JNI critical region

```

From this DTrace output, we can see that the probes `FindClass_entry` and `FindClass_return` have fired due to a JNI function call within a critical region. The output also shows some system calls related to calling `printf()` in the JNI critical region. The native code for this application shows the guilty function:

```

#include "t.h"

/*
 * Class:      t
 * Method:     func
 * Signature:  ([I)V
 */
JNIEXPORT void JNICALL Java_t_func
(JNIEnv *env, jclass clazz, jintArray array) {
    int* value = (int*)env->GetPrimitiveArrayCritical(array, NULL);

    printf("hello world");
    env->FindClass("java/lang/Object");
    env->ReleasePrimitiveArrayCritical(array, value, JNI_ABORT);
}

```

Inspecting Applications with the DTrace jstack Action

Java SE 6 is the first release to contain built-in DTrace probes, but support for the DTrace `jstack` action was

actually first introduced in the Java™ 2 Platform Standard Edition 5.0 Update Release 1. As mentioned in the introduction, the DTrace `jstack` action prints mixed-mode stack traces including both Java method and native function names. As an example of its use, consider the following application, which periodically sleeps to mimic hanging behavior:

```
public class dtest{

    int method3(int stop){
        try{Thread.sleep(500);}
        catch(Exception ex){}
        return stop++;
    }

    int method2(int stop){
        int result = method3(stop);
        return result + 1;
    }

    int method1(int arg){
        int stop=0;
        for(int i=1; i>0; i++){
            if(i>arg){stop=i=1;}
            stop=method2(stop);
        }
        return stop;
    }
    public static void main(String[] args) {
        new dtest().method1(10000);
    }
}
```

To find the cause of the hang, the user would want to know the chain of native and Java method calls in the currently executing thread. The expected chain would be something like:

```
<chain of initial VM functions> ->
dtest.main -> dtest.method1 -> dtest.method2 -> dtest.method3 ->
java/lang/Thread.sleep -> <chain of VM sleep functions> ->
<Kernel pool functions>
```

The following D script uses the DTrace `jstack` action to print the stack trace:

```
Simple D-script: usejstack.d:
#!/usr/sbin/dtrace -s

BEGIN { this->cnt = 0; }

syscall::pollsys:entry
/pid == $1 && tid == 1/
{
    this->cnt++;
    printf("\n\tTID: %d", tid);
    jstack(50);
}

syscall:::entry
/this->cnt == 1/
{
    exit(0);
}
```

And the stack trace itself appears as follows:

```

% usejstack.d 1344 | c++filt
CPU      ID          FUNCTION:NAME
0        316          pollsys:entry
      TID: 1
      libc.so.1`__pollsys+0xa
      libc.so.1`poll+0x52
      libjvm.so`int os_sleep(long long,int)+0xb4
      libjvm.so`int os::sleep(Thread*,long long,int)+0x1ce
      libjvm.so`JVM_Sleep+0x1bc
      java/lang/Thread.sleep
      dtest.method3
      dtest.method2
      dtest.method1
      dtest.main
      StubRoutines (1)
      libjvm.so`void
JavaCalls::call_helper(JavaValue*,methodHandle*,JavaCallArguments*,Thread*)+0x1b5
      libjvm.so`void
os::os_exception_wrapper(void*)(JavaValue*,methodHandle*,JavaCallArguments*,Thread*),Ja
vaValue*,methodHandle*,JavaCallArguments*,Thread*)+0x18
      libjvm.so`void
JavaCalls::call(JavaValue*,methodHandle,JavaCallArguments*,Thread*)+0x2d
      libjvm.so`void
jni_invoke_static(JNIEnv*,JavaValue*,_jobject*,JNICALLType,_jmethodID*,JNI_ArgumentPush
er*,Thread*)+0x214
      libjvm.so`jni_CallStaticVoidMethod+0x244
      java`main+0x642
      StubRoutines (1)

```

The command line shows that the output from this script was piped to the `c++filt` utility, which demangles C++ mangled names making the output easier to read. The DTrace header output shows that the CPU number is 0, the probe number is 316, the thread ID (TID) is 1, and the probe name is `pollsys:entry`, where `pollsys` is the name of the system call. The stack trace frames appear from top to bottom in the following order: two system call frames, three VM frames, five Java method frames, and the remaining frames are VM frames.

It is also worth noting that the DTrace `jstack` action will run on older releases, such as the Java 2 Platform, Standard Edition v 1.4.2, but hexadecimal addresses will appear instead of Java method names. Such addresses are of little use to application developers.

Adding DTrace Probes to Pre-Java SE 6 Releases Using VM Agents

In addition to the `jstack` action, it is also possible for pre-Java SE 6 users to add DTrace probes to their release with the help of *VM Agents*. A VM agent is a shared library that is dynamically loaded into the VM at startup.

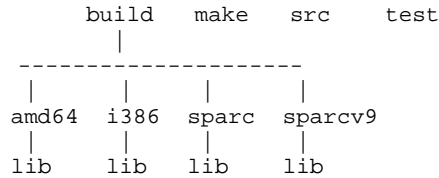
VM agents are available for the following releases:

- For The Java™ 2 Platform, Standard Edition, v 1.4.2, there is a `dvmpi` agent that uses the ¹Java Virtual Machine Profiler Interface (JVMPi).
- For The Java 2 Platform Standard Edition 5.0, there is a `dvmti` agent that uses the JVM™ Tool Interface (JVM TI).

To obtain the agents, visit the DVM java.net project website at <https://solaris10-dtrace-vm-agents.dev.java.net/> and follow the “Documents and Files” link. The file `dvm.zip` contains both binary and source code versions of the agent libraries.

1. The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java™ Platform.

The following diagram shows an abbreviated view of the resulting directory structure once `dvm.zip` has been extracted:



Each `lib` directory contains the pre-built binaries `dvmti.jar`, `libdvmmpi.so`, and `libdvmti.so`. If you prefer to compile the libraries yourself, the included `README` file contains all necessary instructions.

Once unzipped, the VM must be able to find the native libraries on the filesystem. This can be accomplished either by copying the libraries into the release with the other shared libraries, or by using a platform-specific mechanism to help a process find it, such as `LD_LIBRARY_PATH`. In addition, the agent library itself must be able to find all the external symbols that it needs. The `ldd` utility can be used to verify that a native library knows how to find all required externals.

Both agents accept options to limit the probes that are available, and default to the least possible performance impact. To enable the agents for use in your own applications, run the `java` command with one additional option: `-Xrundvmpi` or `-Xrundvmti` (for defaults), or `-Xrundvmpi:all` or `-Xrundvmti:all` (for all probes).

For additional options, consult the DVM agent `README`.

Both agents have their limitations, but `dvmmpi` has more, and we recommend using the Java™ 2 Platform Standard Edition 5.0 Development Kit (JDK 5.0) and the `dvmti` agent if possible.

When using the agent-based approach, keep in mind that:

- The `dvmmpi` agent uses JVMPI and only works with one collector. JVMPI has historically been an unstable, experimental interface, and there is a performance penalty associated with using it. JVMPI only works with JDK™ 5.0 and earlier.
- The `dvmti` agent uses JVM TI and only works with JDK™ 5.0 and later. It works with all collectors, has little performance impact for most probes, and is a formal and much more stable interface.
- Both agents have some performance penalty for method entry/exit and object alloc/free, less so with the `dvmti` agent.
- The `dvmti` agent uses BCI (byte code instrumentation), and therefore adds bytecodes to methods (if method entry/exit or object alloc/free probes are active).
- Enabling the allocation event for the JVM TI agent creates an overhead even when DTrace is not attached, and the JVMPI agent severely impacts performance and limits deployment to the serial collector.

Appendix A provides a D script for testing DVM probes.

The DVM agent provider interface, shown in Appendix B, lists all probes provided by `dvmmpi` and `dvmti`.

The hotspot Provider

This following section lists all probes published by the `hotspot` provider.

VM Lifecycle Probes

Three probes are available related to the VM lifecycle:

vm-init-begin	This probe fires just as the VM initialization begins. It occurs just after JNI_CreateVM() is called, as the VM is initializing.
vm-init-end	This probe fires when the VM initialization finishes, and the VM is ready to start running application code.
vm-shutdown	Probe that fires as the VM is shutting down due to program termination or error

Thread Lifecycle Probes

Two probes are available for tracking thread start and stop events:

thread-start	Probe that fires as a thread is started
thread-stop	Probe that fires when the thread has completed

Each of these probes has the following arguments:

args[0]	A pointer to mUTF-8 string data which contains the thread name
args[1]	The length of the thread name (in bytes)
args[2]	The Java thread ID. This is the value that will match other hotspot probes which contain a thread argument.
args[3]	The native/OS thread ID. This is the ID assigned by the host operating system.
args[4]	A boolean value which indicates if this thread is a daemon or not. A value of 0 indicates a non-daemon thread.

Classloading Probes

Two probes are available for tracking class loading and unloading activity:

class-loaded	Probe that fires after the class has been loaded
class-unloaded	Probe that fires after the class has been unloaded from the system

Each of these probes has the following arguments:

args[0]	A pointer to mUTF-8 string data which contains the name of the class begin loaded
args[1]	The length of the class name (in bytes)
args[2]	The class loader ID, which is a unique identifier for a class loader in the VM. This is the class loader that has loaded or is loading the class
args[3]	A boolean value which indicates if the class is a shared class (if the class was loaded from the shared archive)

Garbage Collection Probes

The following probes measure the duration of a system-wide garbage collection cycle (for those garbage

collectors that have a defined begin and end), and each memory pool can be tracked independently. The probes for individual pools pass the memory manager's name, the pool name, and pool usage information at both the begin and end of pool collection.

The provider's GC-related probes are:

gc-begin	Probe that fires when system-wide collection is about to start. It's one argument (arg[0]) is a boolean value which indicates if this is to be a Full GC
gc-end	Probe that fires when system-wide collection has completed. No arguments.
mem-pool-gc-begin	Probe that fires when an individual memory pool is about to be collected. Provides the arguments listed below
mem-pool-gc-end	Probe that fires after an individual memory pool has been collected. Provides the arguments listed below

Memory pool probe arguments:

args[0]	A pointer to mUTF-8 string data which contains the name of the manager which manages this memory pool
args[1]	The length of the manager name (in bytes)
args[2]	A pointer to mUTF-8 string data which contains the name of the memory pool
args[3]	The length of the memory pool name (in bytes)
args[4]	The initial size of the memory pool (in bytes)
args[5]	The amount of memory in use in the memory pool (in bytes)
args[6]	The number of committed pages in the memory pool
args[7]	The maximum size of the memory pool

Method Compilation Probes

The following probes indicate which methods are being compiled and by which compiler. Then, when the method compilation has completed, it can be loaded and possibly unloaded later. Probes are available to track these events as they occur.

Probes that mark the begin and end of method compilation:

method-compile-begin	Probe that fires as method compilation begins. Provides the arguments listed below
method-compile-end	Probe that fires when method compilation completes. In addition to the arguments listed below, argv[8] is a boolean value which indicates if the compilation was successful

Method compilation probe arguments:

args[0]	A pointer to mUTF-8 string data which contains the name of the compiler which is compiling this method
---------	--

args[1]	The length of the compiler name (in bytes)
args[2]	A pointer to mUTF-8 string data which contains the name of the class of the method being compiled
args[3]	The length of the class name (in bytes)
args[4]	A pointer to mUTF-8 string data which contains the name of the method being compiled
args[5]	The length of the method name (in bytes)
args[6]	A pointer to mUTF-8 string data which contains the signature of the method being compiled
args[7]	The length of the signature(in bytes)

When compiled methods are installed for execution, the following probes are fired:

compiled-method-load	Probe that fires when a compiled method is installed. In addition to the arguments listed below, <code>argv[6]</code> contains a pointer to the compiled code, and <code>argv[7]</code> is the size of the compiled code
compiled-method-unload	Probe that fires when a compiled method is uninstalled. Provides the arguments listed below

Compiled method loading probe arguments:

args[0]	A pointer to mUTF-8 string data which contains the name of the class of the method being installed
args[1]	The length of the class name (in bytes)
args[2]	A pointer to mUTF-8 string data which contains the name of the method being installed
args[3]	The length of the method name (in bytes)
args[4]	A pointer to mUTF-8 string data which contains the signature of the method being installed
args[5]	The length of the signature(in bytes)

Monitor Probes

As an application runs, threads will enter and exit monitors, wait on objects, and perform notifications. Probes are available for all wait and notification events, as well as for contended monitor entry and exit events. A contended monitor entry is the situation where a thread attempts to enter a monitor when another thread is already in the monitor. A contended monitor exit event occurs when a thread leaves a monitor and other threads are waiting to enter to the monitor. Thus, contended enter and contended exit events may not match up to each other in relation to the thread that encounters these events, though it is expected that a contended exit from one thread should match up to a contended enter on another thread (the thread waiting for the monitor).

All monitor events provide the thread ID, a monitor ID, and the type of the class of the object as arguments. It is expected that the thread and the class will help map back to the program, while the monitor ID can provide matching information between probe firings.

Since the existence of these probes in the VM causes performance degradation, they will only fire if the VM has been started with the command-line option `-XX:+ExtendedDTraceProbes`. By default they are present in any listing of the probes in the VM, but are dormant without the flag. It is intended that this restriction be removed in future releases of the VM, where these probes will be enabled all the time with no impact to performance.

The available probes are:

<code>monitor-contended-enter</code>	Probe that fires as a thread attempts to enter a contended monitor
<code>monitor-contended-entered</code>	Probe that fires when the thread successfully enters the contended monitor
<code>monitor-contended-exit</code>	Probe that fires when the thread leaves a monitor and other threads are waiting to enter
<code>monitor-wait</code>	Probe that fires as a thread begins a wait on an object via <code>Object.wait()</code> . The probe has an additional argument, <code>args[4]</code> which is a 'long' value which indicates the timeout being used.
<code>monitor-waited</code>	Probe that fires when the thread completes an <code>Object.wait()</code> and has been either been notified, or timed out
<code>monitor-notify</code>	Probe that fires when a thread calls <code>Object.notify()</code> to notify waiters on a monitor
<code>monitor-notifyAll</code>	Probe that fires when a thread calls <code>Object.notifyAll()</code> to notify waiters on a monitor

Monitor probe arguments:

<code>args[0]</code>	The Java thread identifier for the thread performing the monitor operation
<code>args[1]</code>	A unique, but opaque identifier for the specific monitor that the action is performed upon
<code>args[2]</code>	A pointer to mUTF-8 string data which contains the name of the class of the object being acted upon
<code>args[3]</code>	The length of the class name (in bytes)

Application Tracking Probes

A few probes are provided to allow fine-grained examination of the Java thread execution. These consist of probes that fire anytime a method is entered or returned from, as well as a probe that fires whenever a Java object has been allocated.

Since the existence of these probes in the VM causes performance degradation, they will only fire if the VM has been started with the command-line option `-XX:+ExtendedDTraceProbes`. By default they are present in any listing of the probes in the VM, but are dormant without the flag. It is intended that this restriction be

removed in future releases of the VM, where these probes will be enabled all the time with no impact to performance.

The method entry and return probes:

method-entry	Probe which fires when a method is begin entered. Only fires if the VM was created with the <code>ExtendedDtraceProbes</code> command-line argument.
method-return	Probe which fires when a method returns normally or due to an exception. Only fires if the VM was created with the <code>ExtendedDtraceProbes</code> command-line argument.

Method probe arguments:

args[0]	The Java thread ID of the thread that is entering or leaving the method
args[1]	A pointer to mUTF-8 string data which contains the name of the class of the method
args[2]	The length of the class name (in bytes)
args[3]	A pointer to mUTF-8 string data which contains the name of the method
args[4]	The length of the method name (in bytes)
args[5]	A pointer to mUTF-8 string data which contains the signature of the method
args[6]	The length of the signature(in bytes)

The available allocation probe:

object-alloc	Probe that fires when any object is allocated, provided that the VM was created with the <code>ExtendedDtraceProbes</code> command-line argument.
--------------	---

The object allocation probe has the following arguments:

args[0]	The Java thread ID of the thread that is allocating the object
args[1]	A pointer to mUTF-8 string data which contains the name of the class of the object being allocated
args[2]	The length of the class name (in bytes)
args[3]	The size of the object being allocated

The hotspot_jni Provider

The JNI provides a number of methods for invoking code written in the Java Programming Language, and for examining the state of the VM. DTrace probes are provided at the entry point and return point for each of these methods. The probes are provided by the `hotspot_jni` provider. The name of the probe is the name of the JNI method, appended with `"_entry"` for enter probes, and `"_return"` for return probes. The arguments available at each entry probe are the arguments that were provided to the function (with the exception of the `Invoke*`

methods, which omit the arguments that are passed to Java method). The return probes have the return value of the method as an argument (if available).

Appendix A: dvm_probe_test.d

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

/* #pragma D option quiet */

dvm$1::vm-init
{
    printf("  vm-init");
}

dvm$1::vm-death
{
    printf("  vm-death");
}

dvm$1::thread-start
{
    printf("  tid=%d, thread-start: %s ", tid, copyinstr(arg0));
}

dvm$1::thread-end
{
    printf("  tid=%d, thread-end ", tid);
}

dvm$1::class-load
{
    printf("  tid=%d, class-load: %s ", tid, copyinstr(arg0));
}

dvm$1::class-unload
{
    printf("  tid=%d, class-unload: %s ", tid, copyinstr(arg0));
}

dvm$1::gc-start
{
    printf("  tid=%d, gc-start ", tid);
}

dvm$1::gc-finish
{
    printf("  tid=%d, gc-finish ", tid);
}

dvm$1::gc-stats
{
    printf("  tid=%d, gc-stats: used objects: %ld, used object space: %ld ",
        tid, arg0, arg1);
}

dvm$1::object-alloc
{
    printf("  tid=%d, object-alloc: class name: %s, size: %ld ",
        tid, copyinstr(arg0), arg1);
}

dvm$1::object-free
{
    printf("  tid=%d, object-free: class name: %s ",
        tid, copyinstr(arg0));
}
```

```

dvm$1:::monitor-contended-enter
{
    printf("  tid=%d, monitor-contended-enter:  thread name: %s  ",
           tid, copyinstr(arg0));
}

dvm$1:::monitor-contended-entered
{
    printf("  tid=%d, monitor-contended-entered: thread name: %s  ",
           tid, copyinstr(arg0));
}

dvm$1:::monitor-wait
{
    printf("  tid=%d, monitor-wait:  thread name: %s, time-out: %ld  ",
           tid, copyinstr(arg0), arg1);
}

dvm$1:::monitor-waited
{
    printf("  tid=%d, monitor-waited: thread name: %s, time-out: %ld  ",
           tid, copyinstr(arg0), arg1);
}

dvm$1:::method-entry
{
    printf("  tid=%d, method-entry:  %s:%s %s  ",
           tid, copyinstr(arg0), copyinstr(arg1), copyinstr(arg2));
}

dvm$1:::method-return
{
    printf("  tid=%d, method-return: %s:%s %s  ",
           tid, copyinstr(arg0), copyinstr(arg1), copyinstr(arg2));
}

pid$1:::exit:entry
/execname == "java"/
{
    printf("  tid=%d, D-script exited: pid=%d  \n", tid, pid);
    exit(0);
}

```

Appendix B: DVM Agent Provider Interface

```

provider dvm {
    probe vm__init();
    probe vm__death();
    probe thread__start(char *thread_name);
    probe thread__end();
    probe class__load(char *class_name);
    probe class__unload(char *class_name);
    probe gc__start();
    probe gc__finish();
    probe gc__stats(long used_objects, long used_object_space);
    probe object__alloc(char *class_name, long size);
    probe object__free(char *class_name);
    probe monitor__contended__enter(char *thread_name);
    probe monitor__contended__entered(char *thread_name);
    probe monitor__wait(char *thread_name, long timeout);
    probe monitor__waited(char *thread_name, long timeout);
    probe method__entry(char *class_name, char *method_name, char *method_signature);
    probe method__return(char *class_name, char *method_name, char *method_signature);
};

```

©2005 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA 95054 USA

All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun logo, Java, the Java Coffee Cup logo, Java HotSpot, Solaris, JVM, Java Virtual Machine, Java Development Kit, JDK and Java Native Interface are trademarks, registered trademarks or service marks of Sun Microsystems, Inc. in the United States and other countries.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries.

Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED 'AS IS' AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS HELD TO BE LEGALLY INVALID.

©2005 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA 95054 USA

All rights reserved. Sun, Sun Microsystems, the Sun logo, Java, the Java Coffee Cup logo, Java HotSpot, Solaris, JVM, Java Virtual Machine, Java Development Kit, JDK and Java Native Interface are trademarks, registered trademarks or service marks of Sun Microsystems, Inc. in the United States and other countries.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries.

Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.
