DTrace and Java™ Technology: Taking Observability to the Next Dimension

Jonathan Haslam
Simon Ritter
Sun Microsystems
Presentation Goal

Learn how to use DTrace providers to gain more insight into why your Java application is doing what you've told it to do rather than what you want it to do.

Entertain you with a demonstration of DTrace in 3D (glasses provided)
Agenda

> Introduction to DTrace
> The DVM provider
> The hotspot provider
> JSDT and JDK™ 7
> 3D Visualisation of call graphs
> Summary and Resources
Agenda

> Introduction to DTrace
> The DVM provider
> The hotspot provider
> JSDT and JDK 7
> 3D Visualisation of call graphs
> Summary and Resources
Introduction to DTrace

> Publicly available since Jan 2005.

> Dynamic instrumentation of the whole software system – top-to-bottom, end-to-end
  
  • Application-centric tools tend to have narrow focus and be just for development
  
  • System-centric tools tend to be static and inflexible

> Use on production systems
  
  • completely safe
  
  • zero disabled overhead, minimal enabled overhead
DTrace Terminology

> Probe

- A place of interest in the system where we can make observations

> Provider

- Instruments a particular area of the system
- Makes probes available
- Transfers control into the DTrace framework when an enabled probe is hit.
DTrace Terminology

> Aggregation

- Often, patterns are more interesting than individual datum
- aggregate data together to look for trends
- result of an aggregation function (count(), sum(), max(), avg(), quantize()) associated with arbitrary key in an array

```c
syscall::write:entry
{
   @writes[execname, pid] = sum(arg2);
}
```
jstack() Action

- jstack action prints mixed mode stack trace
- Both Java frames and native (C/C++) frames are shown
- Only JVM versions 5.0_01 and above are supported
- jstack shows hex numbers for JVM versions before 5.0_01
- First (optional) argument limits number of frames shown
- Second (optional) argument changes string size
- jstackstrsize pragma/-x to increase buffer for all jstack calls

```bash
#!/usr/sbin/dtrace -s
syscall::open:entry
/ pid == $1 /
{
    jstack(50, 8192);
}
```
Agenda

> Introduction to DTrace
> The DVM provider
> The hotspot provider
> JSDT and JDK 7
> 3D Visualisation of call graphs
> Summary and Resources
The dvm Provider

> java.net project to add DTrace support in
  • 1.4.2 (libdvmpi.so)
  • 1.5 (libdvmti.so)
  • https://solaris10-dtrace-vm-agents.dev.java.net/
> Download shared libs
> Add location of libs to LD_LIBRARY_PATH variable
> Set JAVA_TOOL_OPTIONS to -Xrundvmti:all
> Name of provider - “dvm”
The dvm Provider: Probes

- dvm probes and their signatures:

  vm-init(), vm-death()
  thread-start(char *thread_name), thread-end()
  class-load(char *class_name)
  class-unload(char *class_name)
  gc-start(), gc-finish()
  gc-stats(long used_objects, long used_object_space)
  object-alloc(char *class_name, long size)
  object-free(char *class_name)
  method__entry(char *class_name, char *method_name,
                char *method_signature)
  method__return(char *class_name, char *method_name,
                 char *method_signature)
Example dvm Provider Usage

dvm$1:::method-entry
/self->in == 0 && copyinstr(arg1) == trace_method/
{
    depth = 1;
    self->in = 1;
    self->caller[depth] = copyinstr(arg1);
    self->timestamp[depth] = vtimestamp;
    @methods[trace_method] = count();
}

dvm$1:::method-return
/self->in && copyinstr(arg1) == trace_method/
{
    total_time = vtimestamp - self->timestamp[1];
    self->in = 0;
    exit(0);
}
Example dvm Provider Usage

dvm$1:::method-entry
/self->in && copyinstr(arg1) == "<init>"/
{
    printf("\t"%s" -> "%s";\n",
            self->caller[depth], copyinstr(arg0));
    depth++;
    self->caller[depth] = copyinstr(arg0);
    @constructors[copyinstr(arg0)] = count();
}

dvm$1:::method-entry
/self->in && copyinstr(arg1) != "<init>"/
{
    printf("\t"%s" -> "%s";\n",
            self->caller[depth], copyinstr(arg1));
    depth++;
    self->caller[depth] = copyinstr(arg1);
    @methods[copyinstr(arg1)] = count();
}
Example dvm Provider Usage

dvm$1:::method-return
/self->in && copyinstr(arg1) == "<init>"/
{
    @times[copyinstr(arg0)] =
    sum(vtimestamp - self->timestamp[depth]);
    self->timestamp[depth] = 0;
    depth--;
}

dvm$1:::method-return
/self->in && copyinstr(arg1) != "<init>"/
{
    @times[copyinstr(arg1)] =
    sum(vtimestamp - self->timestamp[depth]);
    self->timestamp[depth] = 0;
    depth--;
}
Agenda

- Introduction to DTrace
- The DVM provider
- The hotspot provider
- JSDT and JDK 7
- 3D Visualisation of call graphs
- Summary and Resources
DTrace in JDK 6

> JDK 6 supports DTrace “out of the box”

> hotspot provider implements all dvm probes plus extensions:
  - Method compilation (method-compile-begin/end)
  - Compiled method load/unload(compiled-method-load/unload)
  - JNI method probes.
    - DTrace probes as entry and return from each JNI method.

> Strings are now unterminated UTF-8 data
  Use associated length value with copyinstr().
JDK 6 DTrace Usage

> Certain probes are expensive
> Turned off by default
  • object-alloc
  • method-entry, method-return
  • All monitor probes: monitor-wait, etc.
> Use command line flag to enable if required
  • -XX:+ExtendedDTraceProbes (all expensive probes)
  • -XX:+DTraceAllocProbes (selective expensive probes)
  • -XX:+DTraceMethodProbes
  • -XX:+DTraceMonitorProbes
JDK6 hotspot_jni Provider

- Probes for Java Native Interface (JNI)
- Located at entry/return points of all JNI functions
- Probe arguments are same as corresponding JNI function arguments (for _entry probes)
- For XXX_return probes, probe argument is return value
- Examples:

  hotspot_jni$1:::GetPrimitiveArrayCritical_entry
  hotspot_jni$1:::GetPrimitiveArrayCritical_return
Hotspot Probe Demo
Agenda

> Introduction to DTrace
> The DVM provider
> The hotspot provider
> JSDT and JDK 7
> 3D Visualisation of call graphs
> Summary and Resources
Java Statically Defined Tracing

> Included in JDK 7
  • Available in current builds
> JSDT provider
> Insert your own DTrace probes
  • Via Java API
> Lighter weight than existing DVM/hotspot probes
  • More focused
> No impact on application
  • Can be monitored by any DTrace consumer
import com.sun.tracing.*;

public interface MyProvider extends Provider {
    public void start(String name);
    public void finish(String name, int result);
    public void exception(String message);
    public void remove(String type);
}
import com.sun.tracing.*;

public interface MyProvider implements Provider {
    public void start(String name);
    public void finish(String name, int result);
    public void exception(String message);
    public void remove(String type);
}
import com.sun.tracing.*

MyProvider provider;

public static void main(String[] args) {
    ProviderFactory factory =
        ProviderFactory.getDefaultFactory();
    provider =
        factory.createProvider(MyProvider.class);
}

void myMethod() {
    provider.start("myMethod");
}
Using JSDT Probes

```c
#!dtrace -s
#pragma D option quiet

MyProvider$1:::start
{
    printf("Starting method %s\n", copyinstr(arg0));
}
```
Agenda

> Introduction to DTrace
> The DVM provider
> The hotspot provider
> JSDT and JDK 7
> 3D visualisation of call graphs
> Summary and Resources
Why 3D Visualisation

> Ability to visualise data in different way
> Additional information can be displayed
  • Timings, number of calls
  • Source code
> Greater variety of ways to navigate the data
  • There's probably loads more we haven't thought of
> Based on project DAVE
  • 2D call graph visualisation
> 3D game engine written in and for Java
  - High performance
  - Open source

> Scene graph based architecture

> High level API
  - Uses lightweight Java games library (LWJGL)
  - Cross-platform via OpenGL, etc
Development With JMoneyEngine

- Powerful, but not everything is simple
  - Google is your friend!

- Surface wrapping on a box
  - Different surface for each side
  - Default box can only have one surface
  - Need to explicitly build box from 6 surfaces

- Overlaying two images provided some challenges
  - Speed of update
  - Correct position of camera (relative v. absolute)
Anaglyph 3D Rendering

> Use coloured filters for each eye
  • Red/Green, Red/Cyan
> Generate two overlayed images, one in each colour
> Shift camera location to simulate where viewers eyes are
  • Small horizontal shift
> Viewers brain is fooled into seeing 3D image
  • Brighter is better, takes time to adjust
3D Call Graph Demo

(Put your glasses on please)
Agenda

> Introduction to DTrace
> The DVM provider
> The hotspot provider
> JSDT and JDK 7
> 3D Visualisation of call graphs
> Summary and resources
Summary

> DTrace allows arbitrary tracing of events
> Cuts across system boundaries to give complete picture
  • JVM activity
  • Native code activity
> Probes provided for standard events in current JVMs
> JSDT will support developer defined probes
  • Very exciting for deep code analysis
For More Information

> opensolaris.org/os/community/dtrace
  • Documentation
  • Quick reference
  • Cookbook
> docs.sun.com/app/docs/doc/817-6223
> solaris10-dtrace-vm-agents.dev.java.net
> jdk7.dev.java.net
Thank You

Jonathan Haslam
jonathan.haslam@sun.com

Simon Ritter
simon.ritter@sun.com