

Using a Tracing Java™ Virtual Machine to gather data on the behavior of Java programs

Mario Wolczko
Sun Microsystems Laboratories
Mario.Wolczko@Sun.COM

Introduction

The Tracing JVM is a modified Java Virtual Machine which can be used to gather data on the behavior of Java programs.¹

The Tracing JVM is a production-quality JVM in which the interpreter and object management system have been modified so that it can emit data about how objects, stacks, classes, etc., are being used in the execution of a Java application. The data are recorded in trace files which characterize some aspect of the execution behavior. Traces can be analyzed to learn about what the application is doing, used as input to simulators of new JVM implementations, etc. The trace-recording module of the Tracing JVM can be modified or substituted to record data in different ways, or to perform analyses while the JVM is running.

This document describes how to use the Tracing JVM, the format of the traces, and how to modify the trace-recording module. The Tracing VM is a work in progress. The Limitations and Extensions section at the end of this document describes which aspects are incomplete. The Tracing JVM is for research use only and is not a supported product.

1. Sun, Sun Microsystems, the Sun logo, Java-based trademarks and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. SPARC is a trademark or registered trademark of SPARC International, Inc.

What is the Tracing JVM for?

The Tracing JVM is used for recording and analyzing the behavior of a Java application. Because the JVM has been modified to perform tracing, it runs considerably slower than a standard JVM, and hence the Tracing JVM is not useful for analyzing the behavior of JVMs themselves, or of this JVM in particular.

The data recorded by the Tracing JVM are independent of any particular JVM implementation (with minor exceptions; see the section on Limitations). Rather than use the address of an object as an identifier (which would pose problems due to compaction and copying GC), objects are numbered sequentially in order of creation, and are identified by their serial numbers. Similarly, classes and threads are numbered sequentially. No GC events are traced, as these would only be applicable for the GC scheme within the Tracing JVM.

Invoking the Tracing JVM

The Tracing JVM is invoked as `java_t`.

The -Xtrace option

A command-line option enables tracing. The option, `-Xtrace`, takes a single argument specifying the name of a trace configuration file:

```
$ java_t -Xtrace:traceObjs.config Hello
Hello, world!
$
```

The trace configuration file

The trace configuration file contains a list of configuration options, one per line. Most of these options specify that a particular kind of event should be passed to the trace recording module. The form of these options is `event <eventname>` where the valid `<eventnames>` are listed below.

The option `outputfile <filename>` specifies the name of the file into which the trace is written.

Example. To record all class-related events in a file called `classevents.trace`, this configuration file, `classevents.config`, will suffice:

```
outputfile classevents.trace
event DefClass
event DefArray
event ResClass
```

A sample configuration file (`traceAll.config`) enabling all events is provided.

Example

Here's a simple example using the Tracing JVM to determine which classes were loaded and resolved by a program:

```
$ java_t -Xtrace:classevents.config hello
```

(The `traceparser` utility converts a binary trace file into a text format easily processed by text utilities.)

```
$ traceparser -v <classevents.trace # -v = verbose
1 DefClass C thread=1 base=5 offset=0 datum=1 java/lang/Thread
2 DefClass C thread=1 base=0 offset=0 datum=2 java/lang/Object
3 DefClass C thread=1 base=4 offset=0 datum=3 java/lang/Class
...
$ traceparser <classevents.trace |
awk '$2=="DefClass" {c[$7]=$8}
     $2=="DefArray" {c[$7]=c[$5]"["}]
     $2=="ResolveClass" {print c[$7]}'
void
boolean
boolean[]
byte
byte[]
...
```

In this example, we first record all the events relating to class loading and resolution. Then we post-process the trace, first converting it to a text form (using `traceparser`). A simple `awk` script records each class name associated with a `DefClass` or `DefArray` event, and prints out that name upon encountering a subsequent `ResolveClass` event.

The trace recording library

The trace-recording module of the Tracing JVM is encapsulated in the shared library `libtracememsys.so`. Source for this library is provided in `tracememsys.[ch]`, allowing the user to modify the library for his or her own purposes. This section describes the functions that must be implemented in the trace recording library, and when they are invoked.

Initialization

At start-up, if the `-Xtrace` command-line option was specified, the JVM calls the function

```
void initTracing(char *fn);
```

passing, in `fn`, the command-line argument to `-Xtrace`. The default implementation of this function opens the file named by `fn`, and scans it for configuration options. An occurrence of the `outputfile` option with a string argument causes a file named by the string to be created (if necessary) and opened for writing. Each occurrence of an event option selects that event type for recording, by setting to `TRUE` a boolean variable controlling the recording interface (described later).

Termination

Before the JVM exits, it calls the function

```
void endTracing();
```

which should perform appropriate cleanup and summary operations.

Event types

A complete list of event types is named in the macro `EVENT_ITER`, which, when expanded, applies its argument to each event type in turn. These names are used as the names of the event types in a configuration file (see the array `eventNames`), and also in the names of the boolean control variables (`tracing_DefClass`, `tracing_DefArray`, etc.). The `ExprStack` event is included in this list for historical reasons and is not used.

An enumeration, `evtKey_t`, also contains these names as enumeration elements. The values of this enumeration are used to identify event types in the trace file.

Here are the event types currently recognized. The event-recording function interface (described later) specifies the arguments in more detail.

- `DefClass`
A new class of objects has been loaded
- `DefArray`
A new array class has been created
- `NewObject`
A new object has been created and zeroed
- `NewArray`
A new array has been created and zeroed
- `PutField`
An instance variable has been assigned
- `PutStatic`
A static variable has been assigned
- `PutArray`
An element of an array has been assigned
- `PushFrame`
A new stack frame has been created
- `PopFrame`
A stack frame has been reclaimed
- `PutStack`
A local variable or operand stack location has been assigned
- `ExprStack`
Not currently used
- `ConstPool`
A constant pool entry has been resolved to an object
- `GetField`
An instance variable has been accessed
- `GetStatic`
A class variable has been accessed

- **GetStack**
A local variable or operand stack location has been accessed
- **GetArray**
An array element has been accessed
- **ResClass**
A class has been resolved
- **BeginInconsistent, EndInconsistent**
These events come in pairs and bracket regions of execution within a thread during which the JVM may have internal references to objects which are not manifest through the tracing interface (see Sun Microsystems Laboratories Technical Report TR-98-70, “GC Points in a Threaded Environment”, Agesen). A garbage collection simulator should not attempt to collect while a thread is in an inconsistent state, because it may reclaim objects prematurely.
- **AddGlobalRoot, RemoveGlobalRoot, PutGlobalRoot**
These events are used to describe the creation, deletion and assignment to of global root locations internal to the JVM. Each root location is identified by an integer.
- **PushLocalRoot, PopLocalRoots, PutLocalRoot**
These events are used to describe the creation, deletion and assignment to of local root locations on a thread’s stack (in the frames that are internal to the JVM). Each root location is identified by an integer. Each creation of a local root results in a **PushLocalRoot** event. However, many roots may be popped at once; the **PopLocalRoots** event names the local root remaining on the top of the stack of the local roots.
- **AddJNIGlobalRoot, RemoveJNIGlobalRoot, PutJNIGlobalRoot**
These events are used to describe the creation, deletion and assignment to of global root locations within the Java Native Interface (JNI). Each root location is identified by an integer.
- **PushJNILocalRoot, PutJNILocalRoot**
These events describe the creation and assignment to local roots in the JNI. Each of these roots is associated with a Java frame, and is deallocated when the frame is deallocated (see **PopFrame**).
- **GetBytecode**
A bytecode has been fetched and decoded (only the first byte of a multi-byte instruction is given in the event description).

Operand types

The JVM deals in 6 fundamental operand types:

- A pointer to an object. In the tracing module, each object is identified by a unique positive integer, of type `objectID_t`.
- A 32-bit signed integer, `java_int`
- A 32-bit floating-point number, `java_float`
- A pointer to an internal C structure or bytecode return address (`void*`, typedefed to `voidstar`)
- A 64-bit signed integer, `java_long`
- A 64-bit floating-point number, `java_double`

Event-recording functions

Associated with each kind of event is an event-recording function which records the data associated with those events. The signatures of these functions are described in more detail below.

Each call to one of these functions is guarded by a test of the associated control variable:

```
if (tracing_PutField)
    traceMem_PutField_java_int(...);
```

The control variables are initialized to false; the corresponding `event` directive in the configuration file sets it to true.

Variable access event-recording functions

Each kind of variable access (Put/Get of a `Field` [i.e., instance variable], `Static`, `Array` [i.e., array element], `Stack` [i.e., expression stack or local variable]) has six associated C functions which record the data associated with that event, one for each fundamental operand type. The name of each function is of the form `traceMem_<event-kind>_<datum-kind>` where `<event-kind>` is one of the names listed above (e.g., `PutField`) and `<datum-kind>` is one the operand type names listed above (e.g., `java_int`). Each function takes four arguments:

1. The thread ID (`threadID_t`)
2. The underlying structure being modified (an `objectID_t` for instance variable and array accesses, a `classID_t` for static variables, a `JavaFrameID_t` for stack and local variables).
3. An integer identifying the instance variable, array element or static.
4. The datum being read or written.

Example. The function that records a write of a float to an instance variable has this signature:

```
void traceMem_PutField_java_float(
    threadID_t thread,
    objectID_t obj,
    int offset,
    java_float floatVal);
```

The `JavaFrameID_t` in `GetStack` and `PutStack` events is negated when a local variable is being read or written, but is positive when the operand stack is being accessed. The offset corresponds to a local variable offset and stack offset, respectively.

Other event-recording functions

Here are the signatures of the other event-recording functions:

```
void traceMem_NewObject_objectID_t(
    threadID_t thrd_id,
    classID_t base_class,
    int size, /* size (in 32-bit words) */
    objectID_t new_obj)

void traceMem_NewArray_objectID_t(
    threadID_t thrd_id,
```

```
classID_t base_array_class,
int size, /* size (in 32-bit words) */
objectID_t new_obj)

void traceMem_DefClass_voidstar(
threadID_t thrd_id,
classID_t new_class,
int num_statics, /* number of statics in this class */
voidstar class_name) /* (char*) name of class */

void traceMem_DefArray_classID_t(
threadID_t thrd_id,
classID_t elem_class, /* class of array elements */
int ignored,
classID_t new_array_class)

void traceMem_ResClass_classID_t(
threadID_t thrd_id,
classID_t superclass,
int numWords, /* the size of instances (in 32-bit words) */
classID_t resolved_class)

void traceMem_ConstPool_objectID_t(
threadID_t thrd_id,
classID_t class,
int cpIdx, /* the constant pool index */
objectID_t resolved_obj) /* the resolved object */

void traceMem_PushFrame_JavaFrameID_t(
threadID_t thrd_id,
java_int num_locals_in_frame,
int max_stack_size, /* the maximum size of the operand stack,
-1 for native methods */
JavaFrameID_t pushed_frame)

void traceMem_PopFrame_JavaFrameID_t(
threadID_t thrd_id,
java_int unused1,
int unused2,
JavaFrameID_t popped_frame)

void traceMem_BeginInconsistent_java_int(
threadID_t thrd_id,
java_int unused1,
int unused2,
java_int unused3)

void traceMem_EndInconsistent_java_int(
threadID_t thrd_id,
java_int unused1,
int unused2,
java_int unused3)
```

```
void traceMem_AddGlobalRoot_rootID_t(
    threadID_t unused0,
    java_int unused1,
    int unused2,
    rootID_t new_root)

void traceMem_RemoveGlobalRoot_rootID_t(
    threadID_t unused0,
    java_int unused1,
    int unused2,
    rootID_t old_root)

void traceMem_PutGlobalRoot_objectID_t(
    threadID_t thrd_id,
    rootID_t target_root,
    int unused2,
    objectID_t datum)

void traceMem_PushLocalRoot_rootID_t(
    threadID_t thrd_id,
    java_int unused1,
    int unused2,
    rootID_t new_root)

void traceMem_PopLocalRoots_rootID_t(
    threadID_t thrd_id,
    java_int unused1,
    int unused2,
    rootID_t last_unpopped_root)

void traceMem_PutLocalRoot_objectID_t(
    threadID_t thrd_id,
    rootID_t target_root,
    int unused2,
    objectID_t datum)

void traceMem_AddJNIGlobalRoot_rootID_t(
    threadID_t unused0,
    java_int unused1,
    int unused2,
    rootID_t new_root)

void traceMem_RemoveJNIGlobalRoot_rootID_t(
    threadID_t unused0,
    java_int unused1,
    int unused2,
    rootID_t old_root)

void traceMem_PutJNIGlobalRoot_objectID_t(
    threadID_t thrd_id,
    rootID_t target_root,
    int unused,
    objectID_t datum)
```

```
void traceMem_PushJNILocalRoot_rootID_t(  
    threadID_t thrd_id,  
    java_int offset_within_frame_of_root,  
    int unused,  
    JavaFrameID_t base_frame)  
  
void traceMem_PutJNILocalRoot_objectID_t(  
    threadID_t thrd_id,  
    JavaFrameID_t base_frame,  
    int offset_within_frame_of_root,  
    objectID_t datum)  
  
void traceMem_GetBytecode_java_int(  
    threadID_t thrd_id,  
    java_int unused1,  
    int unused2,  
    java_int opcode)
```

The default trace format

The trace file written out by the implementation provided in `tracememsys.c` is a sequence of records. Each record contains the data for a single event. Each record is in this format (exceptions listed below):

1. Event kind (1 byte), a value from the enumeration `evtKey_t`.
2. A datum kind (`char`):
 - 'R' reference to an object (`objectID_t`)
 - 'C' reference to a class (`classID_t`)
 - 'I' 32-bit integer (`java_int`)
 - 'F' 32-bit float (`java_float`)
 - 'P' C pointer, 32-bit (`voidstar`)
 - 'L' 64-bit int (`java_long`)
 - 'D' 64-bit float (`java_float`)
 - 'f' 32-bit Java frame id (`JavaFrameID_t`)
3. Thread ID (`threadID_t`). Threads are numbered consecutively from 1 (there may be occasional gaps in the numbering however, for threads which are created but never used).
4. Base; The second argument to the trace-recording function
5. Offset (`int`, 32-bits); the third argument to the trace-recording function
6. Datum (type indicated by the datum kind, item 2 above); the last argument to the trace-recording function.

Exception: A `DefClass` event has an additional field, the name of the class (a string, terminated with `'\n'`); the base is the number of statics, the datum is the class ID, and the offset field is unused.

The events are accumulated in an internal buffer before being output. A filtering pass removes redundant pairs of BeginInconsistent/EndInconsistent events (see the comment on `filter_buf()` in `tracememsys.c`).

Utilities

Two utilities are also provided:

- `traceparser.[ch]`, which is a C program that converts a trace in the default format to a readable representation (`-v` for human consumption).
- `check-trace`, a `nawk` script that checks a trace, as transformed by `traceparser`, for basic sanity (e.g., objects are created before used, classes are defined before used, etc.) This is not a complete sanity checker; just because a trace passes through `check-trace` without complaint does not mean it is a good trace.

Installing the Tracing JVM

The Tracing JVM is supplied in a set of packages constituting a JDK for the Solaris™ operating environment. These packages can be installed using the `pkgadd` command (root access is required). Solaris 2.5.1, 2.6, or 2.7 for the SPARC™ or Intel architectures is required. Additional OS patches may also be required; these are the same as those required by the Java 2 SDK Production Release (see <http://www.sun.com/solaris/java>).

```
# ls -l SUNX*
total 5
drwxr-xr-x  4 mario          512 Mar 19 11:31 SUNXj2dbg
drwxr-xr-x  4 mario          512 Mar 19 11:30 SUNXj2dev
drwxr-xr-x  4 mario          512 Mar 19 11:30 SUNXj2rt
# pkgadd -d `pwd`
```

The following packages are available:

```
1  SUNXj2dbg      Tracing JDK 1.2 debug binaries
   (sparc) 1.2,REV=1999.03.19.11.29
2  SUNXj2dev      Tracing JDK 1.2 development tools
   (sparc) 1.2,REV=1999.03.19.11.29
3  SUNXj2rt       Tracing JDK 1.2 run time environment
   (sparc) 1.2,REV=1999.03.19.11.29
```

Select package(s) you wish to process (or 'all' to process all packages). (default: all) [?,??,q]: 3

`SUNXj2rt` is the minimal installation; the other packages are optional. If a previous version of the Tracing JVM was installed, it should be first removed (packages `SUNXjv-dem`, `SUNXjvdev`, `SUNXjvman`, `SUNXjvrt`, `SUNXjvthr`).

The packages install files into `/usr/java1.2t`. This will not conflict with other Sun-supplied JDKs. `SUNXj2rt` places the `java_t` wrapper in `/usr/java1.2/jre/bin`; `SUNXj2dev` adds a link in `/usr/java1.2t/bin`. The actual binary is in `/usr/java1.2/jre/bin/{sparc,i386}/native_threads`.

Making a new tracing library

To make a new tracing library, the following commands should be used.

```
cc -c -xO4 -DTRACEMEM tracememsys.c
cc -G -o libtracememsys.so tracememsys.o
```

The new library can be linked into the Tracing JVM at run-time by naming its location in the `LD_LIBRARY_PATH` environment variable.

Example

Consider this problem: build a histogram of the bytecodes used by an application. It is possible to use the standard trace format, recording `GetBytecode` events, to handle this problem in a fashion similar to that used earlier for tracking class resolution. However, this is a relatively slow technique, due to the large amount of data recorded in the trace. A faster technique is to replace the trace-recording module with a custom module that performs only this task. Here is a listing of a suitable implementation:

```
/* tracememsys.c for counting opcodes */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include "tracetypes.h"
#include "tracememsys.h"

#define DEFINE_EVENT_FLAG(ev) bool_t tracing_##ev;

EVENT_ITER(DEFINE_EVENT_FLAG)

static void panic(char *s) {
    fprintf(stderr, "%s\n", s);
    abort(); }

static FILE *trace_file = NULL;

/* called by the JVM when the -trace option is used.
   Arg is the string option to -trace. */

void initTracing(char *fn) {
    /* open a file for the result data */
    trace_file= fopen(fn, "w");
    if (trace_file == NULL)
        panic("Cannot open tracing output file!");
    tracing_GetBytecode = TRUE;
    fprintf(stderr,
        "Trace module: counting opcode frequencies\n");
}

static unsigned long long int opc_count[256];

void traceMem_GetBytecode_java_int(
```

```

        threadID_t thrd_id, java_int unused1,
        int unused2, java_int opcode) {
        if (opcode < 0 || opcode > 255)
            panic("opcode out of range!");
        opc_count[opcode]++;
    }

void endTracing() {
    int i;
    for (i= 0; i < 256; ++i)
        fprintf(trace_file, "%d %llu\n", i, opc_count[i]);
    fclose(trace_file);
}

/* all the remaining functions are empty stubs */
#define DEFINE_EMPTY_TRACE_OP(evtKey, datum_t, ptr_t) \
    void traceMem_##evtKey##_##datum_t( \
        threadID_t thrd_id, ptr_t p, int off, datum_t d) { \
}

TRACE_STACK_OP_ITER(DEFINE_EMPTY_TRACE_OP)
TRACE_FIELD_OP_ITER(DEFINE_EMPTY_TRACE_OP)
TRACE_ARRAY_OP_ITER(DEFINE_EMPTY_TRACE_OP)
TRACE_STATIC_OP_ITER(DEFINE_EMPTY_TRACE_OP)
TRACE_FRAME_OP_ITER(DEFINE_EMPTY_TRACE_OP)
TRACE_ROOT_OP_ITER(DEFINE_EMPTY_TRACE_OP)
TRACE_CLASS_OP_ITER(DEFINE_EMPTY_TRACE_OP)
TRACE_NEW_OP_ITER(DEFINE_EMPTY_TRACE_OP)
TRACE_GC_OP_ITER(DEFINE_EMPTY_TRACE_OP)

```

This module is then compiled as shown above, and linked to `java_t` at run-time (note that we have to bypass the wrapper script in `/usr/java1.2t/jre/bin/java_t`, which sets its own `LD_LIBRARY_PATH`):

```

$ export LD_LIBRARY_PATH=/home/mario/tracing-vm/opcode_count
$ /usr/java1.2t/jre/bin/sparc/native_threads/java_t \
  -Xtrace:opcodes hello
Trace module: counting opcode frequencies
Hello, world!
$ cat opcodes
0 345
1 489
2 676
3 4178
...

```

Limitations and Extensions

Unimplemented events

Currently there are no events which record reads of roots (global or local, JVM or JNI).

- Non-deterministic behavior** Although the Tracing JVM attempts to provide an implementation-independent trace of a Java application's behavior, in one important respect this is not possible. There is a degree of non-determinism in any Java application that uses threads; the timing and order of thread switches varies from implementation to implementation and even run to run.
- GC Consistency** The `BeginInconsistent` and `EndInconsistent` events are artefacts of this particular JVM. They are required by any trace client which attempts to simulate GC, but are not necessarily representative of any other JVM.
- Implementation-dependent behavior** Each Java object has a header whose size and layout is implementation-dependent. Rather than ignore this header, the Tracing JVM traces accesses to the header. Normal objects have two-word headers, so that the first instance variable has offset 2. The first word is a collection of bit fields used for hash codes, GC, etc. The second word is a C pointer to the class structure. Arrays have 3-word headers, the additional word being the number of elements in the array.
- Execution speed** To make the tracing modifications as simple as possible, the Just-In-Time (JIT) compiler has been disabled in the Tracing JVM. This incurs a substantial performance penalty. Also, the overhead of tracing can slow the application by an order of magnitude or more. To obtain the best performance, only the events of interest should be selected in the configuration file.
- Using native methods to control tracing** It is a simple task to add native methods, callable from the Java application, which control the tracing behavior of the JVM. These native methods need merely toggle the boolean variables controlling whether a tracing function is called or not. If you implement this please send the code back to the author; he will incorporate it into a future release.
- Adding new events** At present it is not possible for users to add new events (except those with access to the complete source tree). However, all requests for new event types will be given due consideration by the author.

Recent changes

- Type-indeterminate events** In the latest version (Mar 1999), all stack events have precise type information. Additionally, events for GC (local and global roots, consistency) have been added, as has the bytecode fetch event.

Acknowledgments

I am most indebted to the ExactVM implementation team at Sun (especially the Java Topics group at Sun Labs) for designing and implementing a memory system interface that was easy to overload with tracing operations. Thanks also to Alex Jacoby, who was the first client of the Tracing JVM, to Subramanya Sastry for his feedback and comments, to Timothy Heil for beta testing the latest version, and to all other users who provided feedback.