Maxine-VM Documentation

Release 2.7.0

The Maxine Team of the University of Manchester

Apr 09, 2019

Contents

1 Project Overview **3**

2 Citation **5**

3 Getting Started **7**

4 Features **9**

5 Roadmap **11**

6 Acknowledgements **13**

7 Table of Contents **15**

7.1 Build and Usage Instructions . . . 15
7.2 Status . . . 19
7.3 Developing Maxine on IDEs . . . 21
7.4 Papers and Presentations . . . 25
7.5 The Maxine Project: Contributors . . . 28
7.6 Debugging . . . 31
7.7 Frequently Asked Questions . . . 35
7.8 Glossary of Maxine terminology and concepts . . . 37
7.9 Actors . . . 42
7.10 JDK interoperation . . . 44
7.11 VM Boot Image . . . 48
7.12 Memory Management . . . 55
7.13 Management of Code Dependencies . . . 56
7.14 Code Eviction in the Maxine VM . . . 58
7.15 Object representation in the Maxine VM . . . 60
7.16 Schemes: Interfaces for Maxine VM Configuration . . . 68
7.17 Snippets in the Maxine VM . . . 72
7.18 Stack Walking in the Maxine VM . . . 79
7.19 Threads in the Maxine VM . . . 83
7.20 Type-based Logging . . . 89
7.21 Virtual Machine Level Analysis . . . 94
7.22 VM Operations . . . 108
7.23 VM Tooling interface . . . 112
7.24 The Maxine Inspector . . . 114

8 Indices and tables 175

A next generation, highly productive platform for virtual machine research.

CHAPTER 1

Project Overview

In this era of modern, managed languages we demand ever more from our virtual machines: better performance, more scalability, and support for the latest new languages. Research and experimentation is essential but challenging in the context of mature, complex, production VMs written in multiple languages. The Maxine VM is a next generation platform that establishes a new standard of productivity in this area of research. It is written entirely in Java, completely compatible with modern Java IDEs and the standard JDK, features a modular architecture that permits alternate implementations of subsystems such as GC and compilation to be plugged in, and is accompanied by a dedicated development tool (*the Maxine Inspector*) for debugging and visualizing nearly every aspect of the VM's runtime state.

As of the 2.0 release, September 2013, Maxine is no longer an active project at Oracle Labs. As of the 2.1 release, April 2017, Maxine VM is actively maintained and developed at the University of Manchester.

We believe that Maxine represents the state of the art in a research VM, and actively encourage community involvement. The Maxine sources, including VM, Inspector, and other supporting tools, are Open Source and are licensed under GPL version 2.0. To obtain the code please visit https://github.com/beehive-lab.

CHAPTER 2

Citation

For Maxine VM >= v2.1 please cite: Christos Kotselidis, et al. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), 2017.

For the original Maxine VM please cite: C. Wimmer et al, “Maxine: An approachable virtual machine for, and in, java”, In ACM TACO 2013.

To cite the software itself please use .

CHAPTER 3

Getting Started

- *Download and build Maxine* from source on any of the supported *Supported Platforms*.
- Read the technical report "Maxine: An Approachable Virtual Machine For, and In, Java"
- Send any questions to this mailing list (maxinevm@googlegroups.com).
- Read about the current *status of the VM*.
- Learn more about *the Maxine Inspector*, the companion tool for visualizing internal state and debugging the VM: video introduction, video demos, and written documentation.
- Learn more about *Virtual Machine Level Analysis*, an experimental extension for analysis the behavior of application (and eventually the VM).
- View *publications and presentations* about Maxine.
- Read the *Glossary* and *FAQ*.
- Contact us on the the mailing list (or in private) and tell us about your work.

CHAPTER 4

Features

Some of the features of Maxine that make it a compelling platform for (J)VM research include:

- Nearly all of the code base is written in Java and exploits advanced language features appearing in JDK 5 and beyond: for example annotations, static imports, and generics.
- The VM integrates with openJDK. There's no need to download (and build) other implementations of the standard Java classes.
- The source code supports development in Eclipse, Netbeans or IntelliJ all of which provide excellent support for cross-referencing and browsing the code. It also means that refactoring can be confidently employed to continuously improve the structure of the code.
- *The Maxine Inspector* produces visualizations of nearly every aspect of the VM runtime state, and provides advanced, VM-specific debugging.
- The source code is hosted on GitHub making downloading and collaboration easier.

CHAPTER 5

Roadmap

- Implement JVMCI, Upgrade to latest Graal
- Run Truffle on top of Maxine VM/Graal
- Port MMTk to Maxine VM

CHAPTER 6

Acknowledgements

This documentation is heavily based on the original wiki pages (by Oracle) that can be found here and here.

CHAPTER 7

Table of Contents

7.1 Build and Usage Instructions

The easiest way to build and use Maxine is through the `beehivelab/maxine-dev` docker image. This image comes with all dependencies installed and configured. You only need to get the latest maxine sources and build them through the container.

7.1.1 Prerequisites

The docker image expects the source code to be in the layout shown below:

```
$ tree -L 1 maxine-src
maxine-src
├── maxine
└── mx
```

To get the source code in this layout execute:

```
$ mkdir maxine-src
$ git clone https://github.com/graalvm/mx.git maxine-src/mx
$ git clone https://github.com/beehive-lab/Maxine-VM.git maxine-src/maxine
```

7.1.2 Creating the docker container

From the directory `maxine-src`, that we created in the previous step, run:

```
docker create -u=$(id -u):$(id -g) \
    --mount src="$(pwd)",target="/maxine-src",type=bind \
    --mount src="$HOME/.mx",target="/.mx",type=bind \
    --mount src="/tmp/.X11-unix",target="/tmp/.X11-unix",type=bind \
    -e DISPLAY=unix$DISPLAY --cap-add=SYS_PTRACE \
    --name maxine-dev -ti beehivelab/maxine-dev
```

This will create a container named `maxine-dev`.

- `-u=$(id -u):$(id -g)` instructs docker to write and read files as the current user instead of root which is the default.
- `--mount src="$(pwd)",target="/maxine-src",type=bind` essentially mounts the current directory to the docker container under the */maxine-src* directory. Similarly, *–mount src=”$HOME/.mx”,target=”/.mx”,type=bind* does the same for the *~/.mx* directory. Any changes performed to mounted folders outside the docker container are visible in the container and vice versa.
- `--mount src="/tmp/.X11-unix",target="/tmp/.X11-unix",type=bind` mounts the host X11 socket to the container socket.
- `-e DISPLAY=unix$DISPLAY` passes in the `DISPLAY` environment variable.
- `--cap-add=SYS_PTRACE` enables `ptrace` capability for the container.
- `--name maxine-dev` names the new image so that it can later be referenced (to start it, stop it, attach to it etc.).
- `-ti` instructs docker to create an interactive session with a pseudo-tty, to allow us to interact with the container.

7.1.3 Using the `maxine-dev` container

To use the container issue `docker start -i maxine-dev`. This will start the container and open a `bash` shell in it. From this shell you can build and run maxine.

To exit the shell and stop the container type `Ctrl-D`.

7.1.4 Build

- Enter the maxine source directory:

```
cd $MAXINE_HOME
```

- Compile the source code:

```
mx build
```

Executing `mx build` in the `$MAXINE_HOME` directory compiles the Java source code of Maxine to class files using `javac` (or the Eclipse batch compiler if you use the `-jdt` option) and compiles the native code of Maxine to executable code using your platform's C compiler.

The build process attempts to download some necessary files from the internet. If you are behind a firewall set the `HTTP_PROXY` environment variable appropriately before starting the build.

- Generate the boot image:

```
mx image
```

The `mx image` command is used to generate a boot image. This command runs Maxine on a host JVM to configure a prototype, then compiles its own code and data to create an executable program for the target platform.

7.1.5 Running

With the native substrate and a boot image built, the Maxine VM can now be executed.

The `mx vm` command handles the details of class and library paths and provides an interface similar to the standard java launcher command.

The `mx` script includes a command to run a simple HelloWorld program to verify that the VM is working:

```
mx helloworld
```

Now let's use Maxine to run a more substantial program:

```
mx vm -cp test/bin test.output.GCTest2
```

To launch the VM (or any other command for that matter) without using `mx`, the `-v` option echoes the commands issued by the mx script:

```
mx -v helloworld
```

7.1.6 Profiling

Various profiling tools are available for the Java platform, with varying degrees of overhead. Some tools require VM support and the Maxine VM includes two such tools. The first is a simple sampling based profiler with minimal overhead that is provided in the standard VM image and enabled by the `-Xprof` command line option. The second tool is the *Virtual Machine Level Analysis* (VMA) system that works by instrumenting compiled code. Using VMA requires a custom VM image to be built.

Sampling Profiler

Maxine includes a simple sampling-based profiler. It is enabled with the `-Xprof` command line option. The full syntax for the option is `-Xprof:frequency=f,depth=d,dump=s,flat=t,sort=t,systhreads=t`, where everything after the `-Xprof` is optional. The control arguments have the following interpretation:

- **frequency=f**: Sets the frequency of the samples to `f` milliseconds. The default is 10.
- **depth=d**: Records the stacks of threads at sample points to a depth of `d`. The default is 16.
- **dump=s**: Dumps the accumulated stack traces every s seconds. The default is zero which results in the traces being output only at VM termination.
- **sort=t**: Sorts the stack traces by thread and sample counts if t is true. The default value is true unless dump is non-zero, as the sorting incurs both CPU and allocation overhead. In unsorted mode the stack traces are output in an arbitrary order, each followed by the list of threads and sample counts for that trace. In sorted mode, the traces for each thread are output separately, with the traces ordered from highest to lowest sample count.
- **flat=t**: If t is true, the output is sorted and, for each sample, only the method at the top of the stack is listed. Therefore, this option also implies `depth=1`. The default value is `true`.
- **systhreads=t**: Include system (VM) threads in the analysis if `t` is `true`. The default is false.

If the `=t` in the truth-valued options is omitted, it is the same as `t=true`.

The profiler is implemented as a separate thread that wakes up periodically, based on the given frequency (slightly randomized), stops all threads and records their stack traces. Since threads only stop at safepoints there is some inevitable inaccuracy in the reported trace. In particular, a hot method that contains no loops will not appear in the output. However, the stack trace will likely show the closest caller that contains a loop (or a system call that will cause the thread to reach a safepoint).

The data is output using the Maxine log mechanism, so can be captured in a file by setting the `MAXINE_LOG_FILE` environment variable.

7.1.7 Building Maxine without docker

Dependencies

Maxine depends on the MX tool for its build process. To get it and add it to your `PATH` execute:

```
sudo apt-get install python2.7           # MX depends on python 2.7
mkdir -p $WORKDIR
cd $WORKDIR
git clone https://github.com/graalvm/mx
export PATH=$PATH:$(pwd)/mx
```

Maxine also depends on openJDK 8. To get it from the ubuntu repositories run:

```
sudo apt-get install openjdk-8-jdk
```

Maxine is open source software, licensed under the GPL version 2.0 and is hosted on GitHub. Since Maxine is hosted in a git repository we need to install `git` as well:

```
sudo apt-get install git
```

Environment variables

To build maxine natively we first need to define a number of environment variables:

1. Define the directory you want to work in:

   ```
   export WORKDIR=/path/to/workdir
   ```

2. Define the JDK to be used:

   ```
   export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
   ```

3. Define `MAXINE_HOME`:

   ```
   export MAXINE_HOME=$WORKDIR/maxine
   ```

4. Optionally (needed to run `maxvm` binary directly):

   - Extend `PATH` to include the *to be generated* `maxvm`:

     ```
     export PATH=$PATH:$MAXINE_HOME/com.oracle.max.vm.native/generated/
     ↪linux/
     ```

   - Define `LD_LIBRARY_PATH`:

```
export LD_LIBRARY_PATH=$MAXINE_HOME/com.oracle.max.vm.native/
↪generated/linux/
```

Get the source code

1. Make sure the project directory exists and enter it:

```
mkdir -p $WORKDIR
cd $WORKDIR
```

2. Get the Maxine VM source code:

```
git clone https://github.com/beehive-lab/Maxine-VM.git maxine
```

This command will create a directory named `maxine` with the contents checked out from the git repository.

7.1.8 Choice of Optimizing Compiler

Maxine provides two optimizing compilers, C1X and Graal. The former, an evolution of the Hostpot client compiler, is very stable but no longer under development. Graal is more akin to the Hotspot server compiler and is under active development and improvement. The default image build still uses C1X as the optimizing compiler, but it is possible to select Graal, both for runtime compilations and for compiling the VM boot image (the latter is currently unstable). To build a boot image with Graal as the runtime optimizing compiler, use the following command:

```
mx image @c1xgraal
```

In this case the optimizing compiler is actually a hybrid of C1X and Graal, with C1X being used as a fallback option if the Graal compilation fails. Note that the VM boot image is considerably larger (~100MB) with Graal included.

To compile the boot image itself with Graal, do:

```
mx image @c1xgraal-boot
```

The Graal-compiled VM boot image will execute a few simple test programs but currently is not robust enough to be the default.

7.2 Status

7.2.1 Supported Platforms

Maxine is being developed and tested on the following configurations:

Architecture	OS	Java	MaxineVM Version
X86_64	Ubuntu 18.04	OpenJDK 8 (u191)	2.7.0
x86_64	macOS Mojave 10.14.3	OpenJDK 8 (u181)	2.7.0
Aarch64	Ubuntu 16.04/18.04	OpenJDK 8 (u181)	2.6.0
ARMv7	Ubuntu 16.04	OpenJDK 7 u151	2.4.0

MaxineVM - JDK version compatibility table

The table below shows the JDK version required to build each version of MaxineVM.

MaxineVM Version	Java Version
>= 2.7.0	Open JDK 8 u191
2.5.1 - 2.6.0	Open JDK 8 u181
2.4.0 - 2.5.0	Open JDK 7 or 8 u151
2.2 - 2.3.0	Open JDK 7 or 8 u151
2.1.1	Open JDK 7 u131
2.0 - 2.1.0	Oracle JDK 7 u25
< 2.0	Oracle JDK 7 u6

To get OpenJDK 7 u151 in Ubuntu 16.04 on x86 you can use the following debian packages:

```
cd /tmp

export ARCH=amd64                          # or arm64
export JAVA_VERSION=7u151-2.6.11-3         # or 8u151-b12-1
export JAVA=openjdk-7                      # or openjdk-8
export FCONFIG_VERSION=2.12.3-0.2
export BASE_URL=http://snapshot.debian.org/archive/debian/20171124T100538Z

for package in jre jre-headless jdk dbg; do
wget ${BASE_URL}/pool/main/o/${JAVA}/${JAVA}-${package}_${JAVA_VERSION}_$
↪{ARCH}.deb
done

for package in fontconfig-config libfontconfig1; do
wget ${BASE_URL}/pool/main/f/fontconfig/${package}_${FCONFIG_VERSION}_all.
↪deb
done

wget http://ftp.uk.debian.org/debian/pool/main/libj/libjpeg-turbo/
↪libjpeg62-turbo_1.5.1-2_${ARCH}.deb

sudo dpkg -i ${JAVA}-jdk_${JAVA_VERSION}_${ARCH}.deb ${JAVA}-jre_${JAVA_
↪VERSION}_${ARCH}.deb ${JAVA}-jre-headless_${JAVA_VERSION}_${ARCH}.deb $
↪{JAVA}-dbg_${JAVA_VERSION}_${ARCH}.deb libjpeg62-turbo_1.5.1-2_${ARCH}.
↪deb fontconfig-config_${FCONFIG_VERSION}_all.deb libfontconfig1_$
↪{FCONFIG_VERSION}_all.deb
sudo apt-get install -f
```

7.2.2 Maturity

Maxine VM is being tested against the SPECjvm2008 and DaCapo-9.12-bach benchmark suites. The following tables show the status of each benchmark on each supported platform.

SpecJVM2008

Benchmark	ARMv7	AArch64	X86 C1X	X86 C1X-Graal
startup	PASS	PASS	PASS	PASS
compiler	PASS	PASS	PASS	FAIL
compress	PASS	PASS	PASS	PASS
crypto	PASS	PASS	PASS	PASS
derby	FAIL	FAIL	PASS	FAIL
scimark	PASS	PASS	PASS	PASS
serial	PASS	PASS	PASS	
sunflow	FAIL	FAIL	PASS	FAIL
xml	FAIL	PASS	PASS	PASS
pass-rate	90%	92%	100%	55%

Note: The pass-rate is calculated based on the individual tests of each group, e.g., compiler contains 2 tests while serial only 1. As a result, groups have different weights.

DaCapo-9.12-bach

Benchmark	ARMv7	AArch64	X86 C1X	X86 C1X-Graal
avrora	PASS	PASS	PASS	PASS
batik	FAIL	FAIL	FAIL	FAIL
eclipse	FAIL	FAIL	PASS	FAIL
fop	PASS	PASS	PASS	PASS
h2	PASS	PASS	PASS	PASS
jython	PASS	PASS	PASS	PASS
luindex	PASS	PASS	PASS	PASS
lusearch	PASS	PASS	PASS	PASS
pmd	FAIL	PASS	PASS	PASS
sunflow	PASS	PASS	PASS	PASS
tomcat	FAIL	PASS	PASS	PASS
tradebeans	FAIL	FAIL	PASS	PASS
tradesoap	FAIL	FAIL	PASS	PASS
xalan	PASS	PASS	PASS	PASS
pass-rate	62%	77%	100%	92%

Note: batik fails due to a library that is not available on openJDK, it is thus omitted from the pass-rate.

7.2.3 Issues

Any issues are reported in the issue tracker.

7.3 Developing Maxine on IDEs

7.3.1 Eclipse

Launch Configuration

Once you have installed Eclipse, we recommend modifying its launch configuration by editing the `eclipse.ini` file in the Eclipse installation directory. In particular, you will want to give it more memory with the following options:

```
-XX:MaxPermSize=512m
-Xmx1g
-Xms512m
```

On GNU/Linux the `eclipse.ini` file is located under `/usr/lib/eclipse/`.

On Mac OS X the `eclipse.ini` file is hidden inside the application bundle Eclipse.app, which shows up as a single executable in the Finder. Open the bundle, either with the Finder *Show Package Contents* contextual menu item, or with an editor such as Emacs. The file can be found at: `Eclipse.app/Contents/MacOS/eclipse.ini`.

Configuring Eclipse

When first launching Eclipse for Maxine development, you should create a new workspace in the directory where you have checked out the Maxine sources.

You will then need to install the CDT plugin if you want to edit and build the Maxine C code from within Eclipse. We also strongly recommend installing the Checkstyle plugin to simplify conforming with the Maxine coding conventions.

Next, ensure that the default JRE being used for development is at least a JDK 7 installation. This will be the case by default when Eclipse is running on JDK 7. Otherwise you will manually have to change this setting in the `Java > Installed JREs` preference page. Once a JDK has been selected, you should set some default VM options for it by selecting it and hitting the `Edit...` button. In the `Default VM Arguments` field in the dialog that comes up, add the `-ea` option. Also add the `-d64` option if you are on a platform (such as Linux or Solaris) where the JVM can be launched in either 32-bit or 64-bit mode. Obviously this requirement will change should there ever be a 32-bit Maxine port.

Creating and Importing the Maxine Eclipse projects

Once Eclipse has been configured, you need to run a short `mx` command that will create the Eclipse project configurations for all the projects in the Maxine workspace:

```
mx ideinit
```

The above command will actually create IDE project configurations for all supported IDEs (currently Eclipse and NetBeans). To create only Eclipse project configurations, replace ideinit with eclipseinit.

You use the Import Wizard to import the created/updated projects.

1. From the main menu bar, select `File > Import...` to open the Import Wizard.
2. Select `General > Existing Project into Workspace` and click `Next`.
3. Choose `Select root directory` and click the associated `Browse` button to locate the top level Maxine directory containing the projects.
4. Under `Projects` select all the projects.

5. Click `Finish` to complete the import.

Once the sources have finished importing, Eclipse will automatically compile them. It will also run `gmake` to build the C code in the Native project. If the latter process appears to fail, the most common cause is `gmake` not being on the `PATH` of the environment from which Eclipse was launched.

Note: Occasionally, a new Eclipse project is added to the Maxine source code. This usually results in an Eclipse error message indicating that a project is missing another required Java project. To handle this, you simply need to repeat the steps above for discovering and importing projects.

Building the boot image

Assuming all the sources compiled successfully, you can build the VM boot image by following the instructions *here*.

To run the VM, open a console and use `mx vm`.

Cloning Git workspaces

Git makes it very easy to clone an existing workspace for the purpose of experimentation. Eclipse on the other hand does not have a simple mechanism for copying settings from one Eclipse workspace to another. We've found that the simplest thing to do is to copy the .metadata directory from an existing Eclipse workspace to the workspace created by the git clone operation.

For example:

```
% ls
maxine
% git clone maxine sandbox
% cp -r maxine/.metadata sandbox/.metadata
```

Then select all the projects (in the Package Explorer view) and perform `File > Refresh` in Eclipse for the cloned workspace.

7.3.2 Netbeans

Ensure that you select a JDK 7 during the installation process. The result of the installation process is a directory named netbeans (hereafter referred to as `$IDE_HOME`). Note that on Mac OS X, the installation directory will be `/Applications/NetBeans` and the directory denoted by `$IDE_HOME` in these instructions is `/Applications/NetBeans/NetBeans<version>.app/Contents/Resources/NetBeans`.

Before starting NetBeans, it's useful to tune its configuration by editing the `$IDE_HOME/etc/netbeans.conf` file. In particular, the `netbeans_default_options` value can be modified to increase the heap size of the JVM running NetBeans (e.g. add `-J-Xmx1g` to the value).

Generate the NetBean configuration files

The mx script can be used to generate NetBean project configurations for each project in the Maxine code base. Simply run `mx netbeansinit` and follow the instructions it prints out to the console.

7.3.3 IntelliJ

Generate eclipse project files describing module dependencies

Executing the following command will create eclipse and netbeans project files describing the dependencies between the different modules of Maxine. These project files will later be parsed by IntelliJ to understand and import the module dependencies.

```
mx ideinit
```

Create a new IntelliJ project

Open IntelliJ and:

1. Select `File > New Project`.
2. Select the `Create new Java project from existing sources` option.
3. Use Maxine as the name of the project and for `Project file location`, select the directory where you checked out the Maxine code. When you click `Next`, IntelliJ should find the source directories automatically.
4. IntelliJ will not find any libraries for the project, click `Next`.
5. IntelliJ should infer correct modules for the project, click `Next`.
6. IntelliJ should not infer any facets for the project, click `Finish`.

Add JUnit4 library

You will need JUnit 4.0+ in order to compile Maxine. It is probably best not try to compile Maxine from within IntelliJ before this step; its caches may become confused later, and it won't work anyway.

1. Select `File > Settings`.
2. Select `Project Settings`.
3. Select `Libraries`.
4. Click the plus icon to add a new library.
5. Use the name *JUnit4* for the library.
6. Apply the library to all the modules.
7. Click `Add Classes` and navigate to the location of your junit4.jar file.
8. Click `OK`.

More memory for Java Compiler

Maxine has some rather large source files, and javac will likely run out of memory. You need to increase the amount of memory available to it by:

1. Select `File > Settings`.
2. Select `Compiler`.

3. Change the value for `Maximum heap size` to 1024.

7.4 Papers and Presentations

This page lists some of the papers, presentations, articles and demos that have resulted from the Maxine project and its predecessors.

- Po-An Tsai and Daniel Sanchez. **Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy.** In the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.
- Po-An Tsai, Yee Ling Gan, and Daniel Sanchez. **Rethinking the Memory Hierarchy for Modern Languages.** In 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018.
- Yee Ling Gan. **Redesigning the Memory Hierarchy for Memory-Safe Programming Languages.** Masters Thesis, Massachusetts Institute of Technology, 2018.
- Thomas Shull, Jian Huang, and Josep Torrellas. **Defining a high-level programming model for emerging NVRAM technologies.** In Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang) 2018.
- Foivos S. Zakkak, Andrew Nisbet, John Mawer, Timothy Hartley, Nikolaos Foutris, Orion Papadakis, Andreas Andronikakis, Iain Apreotesei, Christos Kotselidis. **On the Future of Research VMs: A Hardware/Software Perspective.** In Workshop on Modern Language Runtimes, Ecosystems, and VMs (MoreVMs), 2018.
- Foivos S. Zakkak, Juan Fumero, Christos Kotselidis. **Enabling RISC-V support on MaxineVM (poster session).** In the 2018 RISC-V Workshop, May 2018.
- Christos Kotselidis, Andy Nisbet, Foivos S. Zakkak, and Nikos Foutris. **Cross-ISA debugging in meta-circular VMs.** In the 9th International Workshop on Virtual Machines and Intermediate Languages (VMIL '17), October 2017.
- Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, Mikel Luján. **MaxSim: A Simulator Platform for Managed Applications.** In 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2017. **Best paper award**.
- Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, Mikel Luján. **Type Information Elimination from Objects on Architectures with Tagged Pointers Support.** In IEEE Transactions on Computers (TOC), 2017.
- Christos Kotselidis, Andy Nisbet, Foivos S. Zakkak, Nikos Foutris. **Cross-ISA debugging in meta-circular VMs.** In 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL), 2017.
- Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. **Heterogeneous Managed Runtime Systems: A Computer Vision Case Study.** In 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), 2017.
- Christos Kotselidis, Andrey Rodchenko, Colin Barrett, Andy Nisbet, John Mawer, Will Toms, James Clarkson, Cosmin Gorgovan, Amanieu d'Antras, Yaman Cakmakci, Thanos Stratikopoulos, Sebastian Werner, Jim Garside, Javier Navaridas, Antoniu Pop, John Goodacre and Mikel

Lujan. **Project Beehive: A Hardware/Software Co-designed Stack for Runtime and Architectural Research.** In the 9th International Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG'16), January 2016.

- Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, Douglas Simon. **Maxine: An Approachable Virtual Machine For, and In, Java.** In ACM Transactions on Architecture and Code Optimization, volume 9, issue 4, article 30. ACM Press, 2013.doi:10.1145/2400682.2400689.
- Sameer Kulkarni, John Cavazos, Christian Wimmer, Douglas Simon. **Construction of Inlining Heuristics using Machine Learning.** In Proceedings of the International Symposium on Code Generation and Optimization. IEEE, 2013.
- Mohammad Mahdi Shahabi, **Dynamic Location-Based Analysis of Access Contracts**, Masters Thesis, Albert-Ludwigs-Universität Freiburg im Breisgau, August 9, 2012.
- Vinicius H. S. Durelli, Jeff Offutt, and Marcio E. Delamaro. 2012. **Toward Harnessing High-Level Language Virtual Machines for Further Speeding Up Weak Mutation Testing**. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12). IEEE Computer Society, Washington, DC, USA, 681-690.
- Christian Wimmer, Laurent Daynes. **Maxine: A Virtual Machine For, and In, Java.** ECOOP Summer School, Jun 15, 2012.
- Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. 2012. **Fine-grained modularity and reuse of virtual machine components.** In Proceedings of the 11th annual international conference on Aspect-oriented Software Development (AOSD '12). ACM, New York, NY, USA, 203-214.
- Michael Haupt, **Machine Code Management in the Maxine Research Virtual Machine** (Presentation), Hasso-Plattner-Institut Potsdam, Jan 27, 2012
- Michael Haupt, Stefan Marr, Robert Hirschfeld, **CSOM/PL: A Virtual Machine Product Line**, Journal of Object Technology, Vol. 10, 2011
- Michael Haupt, **The Maxine Virtual Machine** (Presentation), Universität Leipzig, Nov 10, 2011; Technische Universität Dortmund, Nov 24, 2011
- Thomas Würthinger, **Extending the Graal Compiler to Optimize Libraries** (Demonstration), SPLASH'11, Portland, OR, October 22-27, 2011
- Michael Bebenita, **Trace-Based Compilation and Optimization in Meta-Circular Virtual Execution Environments**, Ph.D. Dissertation, UC Irvine, 2011
- Victor Luchangco and Virendra J. Marathe, **Revisiting Condition Variables and Transactions**, TRANSACT'11, San Jose, California, June.
- Johannes Eickhold, Markus Knauer, **Sovereign: Migrating Java Threads to Improve Availability of Web Applications**, EclipseCon 2011 (Presentation and Demo), Santa Clara, California, March 21-24, 2011.
- Michael Van De Vanter, **The Maxine Virtual Machine and Inspector: a highly approachable environment for VM Research** (Presentation) San Francisco State University, March 16, 2011, California Polytechnic State University, April 7, 2011
- Victor Luchangco, Virendra J. Marathe, **Transaction Communicators: Enabling Cooperation Among Concurrent Transactions**, PPoPP 2011, 16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, San Antonio, Texas, February 12-16, 2011

- Doug Simon, **What a meta-circular JVM buys you - and what not** (Keynote Talk), PPPJ '10, 8th international Conference on the Principles and Practice of Programming in Java, Vienna, Austria, September 15 - 17, 2010.
- Li, Jianyuan, Jan Simon Rellermeyer, Adrian Schüpbach, and Gustavo Alonso. **Performance analysis and improvement of guestVM for running OSGi-based MacroComponents.** PhD dissertation, ETH Zurich, Department of Computer Science, Systems Group, August 22, 2010
- Johannes Eickhold, **Some introductory short articles about Maxine**
- Doug Simon, **Maxine Adapters** (Sample execution of a Maxine adapter), 2010
- M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, M. Franz, **Trace-based compilation in execution environments without interpreters**, In Proceedings of the 8th international Conference on the Principles and Practice of Programming in Java (Vienna, Austria, September 15 - 17, 2010). PPPJ '10. ACM, New York, NY, 59-68.
- Michael Van De Vanter, **The Maxine Inspector: A Specialized Tool for VM Development** (Presentation and Demo), 2010 JVM Language Summit, July 28, 2010, Santa Clara, California., (video recording)
- Ben Titzer, Thomas Würthinger, Doug Simon, and Marcelo Cintra, **Improving compiler-runtime separation with XIR**, VEE 2010, Proceedings of the International Conference on Virtual Execution Environments, pages 39–50. ACM Press,, SIGPLAN Notices 45, 7 (Jul. 2010), 39-50 (paper) Discusses C1X and XIR. The results in this paper are based on revision 3254 of the Maxine sources.
- Doug Simon, Ben Titzer, **Splicing Modules with a Metacircular Saw: "Snippets" in the Maxine VM** (Invited Talk), VMIL 2009, The 3rd workshop on Virtual Machines and Intermediate Languages, Orlando, Florida October 25, 2009.
- Doug Simon, **the Maxine Research Virtual Machine** (Internet radio interview), Software Engineering Radio, Episode 144 , September 7, 2009.
- Allan Raundahl Gregersen, Douglas Simon and Bo Norregaard Jorgensen, **Towards a Dynamic-Update-Enabled JVM**, 6th ECOOP'2009 Workshop on Reflection, AOP and Meta-Data for Software Evolution, Genova, Italy, July 7, 2009.
- Thomas Wuerthinger, Michael Van De Vanter, Doug Simon, **Multi-Level Virtual Machine Debugging using the Java Platform Debugger Architecture**, Seventh International Andrei Ershov Memory Conference "Perspectives of System Informatics", Novosibirsk, Russia, 15-19 June, 2009.
- Ben Titzer, **The Maxine Virtual Machine** (Presentation), Brown IPP Symposium on Standardizing Transactional Memory, Brown University, April 20, 2009.
- Bernd Mathiske, **Leveraging Meta-Circularity in the Maxine VM**, (Presentation) 2008 JVM Language Summit, September 24, 2008, Santa Clara, California.,
- Bernd Mathiske, **Systems programming in the Maxine VM: how to enable it and how to get around it**, (Invited Talk), PPPJ08 Principles and Practice of Programming in Java, Modena, Italy, September 9-11, 2008 (video recording)
- Bernd Mathiske, **Systems Programming in the Maxine VM: how to enable it and how to get around it**, 2008 JavaOne conference, San Francisco, California, June 2008
- Bernd Mathiske, Doug Simon, David Ungar, **An assembler and disassembler framework for Java™ programmers**, Science of Computer Programming, Vol. 70, Issues 2-3, February 2008, pp 127-148.

- Bernd Mathiske. 2008. The maxine virtual machine and inspector. In Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA Companion '08). ACM, New York, NY, USA, 739-740.
- Bernd Mathiske, Doug Simon, David Ungar, **The Project Maxwell assembler system.** In Proceedings of the 4th international Symposium on Principles and Practice of Programming in Java (Mannheim, Germany, August 30 - September 01, 2006). PPPJ '06, vol. 178. ACM, New York, NY, 3-12.
- David Ungar, Adam Spitz, Alex Ausch, **Constructing a metacircular Virtual machine in an exploratory programming environment**, In Companion To the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (San Diego, California, USA, October 16 - 20, 2005). OOPSLA '05. ACM, New York, NY, 11-20.

7.5 The Maxine Project: Contributors

This page lists the people who have been associated with the Maxine project. For code contribution statistics see github contributions.

7.5.1 The University of Manchester

- Andreas Andronikakis
- Andrey Rodchenko
- Andy Nisbet
- Benjamin Bell
- Christos Kotselidis
- Costas Lebesis
- Florin Blanaru
- Foivos Zakkak
- Iain Apreotesei
- Mikel Lujan
- Orion Papadakis
- Tim Hartley

7.5.2 Oracle Labs

Oracle Labs members who contributed to Maxine:

- Laurent Daynes
- Michael Haupt
- Michael Van De Vanter
- Mick Jordan

- Doug Simon
- Michael Van De Vanter
- Christian Wimmer
- Thomas Würthinger
- Ben Titzer
- Paul Caprioli
- Bernd Mathiske (Principal Investigator, 2005 - 2008)
- Greg Wright (2003 - 2004)

Interns

Maxine would not be where it is today without the valuable input of these interns:

Name	Affiliation	When	Topic
Christian Häubl	Johannes Kepler U. Linz	winter 2012	Profile feedback for Graal
Arian Treffer	HPI Potsdam	summer 2011	object models for dynamic languages
Tobias Pape	HPI Potsdam	summer 2011	execution models for dynamic languages
Gilles Duboscq	Johannes Kepler U. Linz	summer 2011	Graal
Sameer Kulkarni	U. Delaware	summer 2011	machine learning for code optimization
Du Li	U. Nebraska Lincoln	fall 2010	VM support for analysis
Lukas Stadler	Johannes Kepler U. Linz	summer 2010	C1X Hotspot integration
Michael Duller	ETH Zürich	summer 2010	De-opt
Puneet Lakhina	UC Santa Barbara	summer 2010	Maxine Virtual Edition
Thomas Würthinger	Johannes Kepler U. Linz	summer 2009	C1X and XIR
Marcelo Cintra	UC Irvine	summer 2009	interpreter and verifier for IR of C1X
Hannes Payer	U. Salzburg	summer 2009	safepoint synchronization, relocatable watchpoints and GC support in the The Maxine Inspector, immortal memory, TLABs
Michael Bebenita	UC Irvine	summer 2008	trace compilation
Abdulaziz Ghuloum	Indiana U. Bloomington	summer 2008	performance analysis, compiler optimizations
Yi Guo	Rice	summer 2008	performance analysis, compiler optimizations
Christos Kotselidis	U. Manchester	summer 2008	generational garbage collection (Beltway)
Karthik Manivannan	UC Irvine	summer 2008	generational garbage collection (Beltway)
Thomas Würthinger	Johannes Kepler U. Linz	summer 2008	IR visualization, Inspector GUI
Aritra Bandyopadhyay	Colorado State U.	summer 2008	IR visualization, Inspector GUI, array bounds checking
Simon Wilkinson	U. Manchester	spring 2008	multiple modal object monitor implementations for thread synchronization, including biased-locking
Sunil Soman	UC Santa Barbara	winter 2007/2008	safepoint mechanism debugging, semispace GC
Athul Acharya	Purdue	summer 2007	remote interpretor for the The Maxine Inspector

7.6 Debugging

7.6.1 Inspector

The Inspector is the tool co-developed with the VM for debugging. Launching the Inspector is as simple as using the `mx inspect` command.

```
mx inspect -cp com.oracle.max.tests/bin test.output.GCTest2
```

The Inspector window should appear in a few moments. Go *here* for other ways of launching the Inspector.

7.6.2 Testing Maxine

Benchmarks

The most useful way to test Maxine is to execute some of the standard benchmarks on the image previously built with the `mx image` command. In this example, we will use the SpecJVM98 and DaCapo benchmarks. After downloading the benchmarks, set the following environment variables:

```
export SPECJVM98_ZIP=/Users/acme/benchmarks/specjvm98.zip
export DACAPOBACH_JAR=/Users/acme/benchmarks/dacapo-9.12-bach.jar
```

Then execute the following command:

```
mx test -insitu -tests=specjvm98,dacapobach
------------------------------------------------------------------------------
 ↪-------------
Running reference       SpecJVM98 _201_compress:                1607 ms
Running maxvm (std)     SpecJVM98 _201_compress:                3309 ms
 ↪       2.059x
------------------------------------------------------------------------------
 ↪-------------
...
Running reference       DaCapo-bach avrora:                     3960 ms
Running maxvm (std)     DaCapo-bach avrora:                    13815 ms
 ↪       3.488x
------------------------------------------------------------------------------
 ↪-------------
...
```

Note that the harness used to run the benchmarks against a reference VM is a little brittle. In particular, it compares the output written to `stdout` by the two executions and if they don't match, it determines the Maxine VM execution to have failed. We've noticed that for some benchmarks on some platforms, the execution output is not such a reliable fingerprint. To see the actual output of the benchmark, and the exact command used to run it, look in the `maxine-tester/insitu` directory.

Regression testing

Maxine includes a set of tests that are useful in catching regressions and measuring progress when making changes. The three types of tests included in the distribution are described below.

JUnit tests

Tests the Maxine code base prior to building and running the VM. For example, there are JUnit tests for the general purpose utility classes in the Base project. There are also JUnit tests that use the various IR interpreters to ensure that each level of IR in Maxine's compiler produces the correct output.

VM tests

These are tests that are executed on the VM. The first subcategory of VM tests are very simple unit tests that test a specific VM feature and/or Java bytecode instruction in isolation (i.e. avoiding as many other VM features as possible). To ensure strong isolation, these tests are built into the boot image and executed in such a way that precludes using a more general testing framework such as JUnit. The second subcategory of VM tests are called output tests. These tests are comprised of standard Java programs (i.e. they have a class containing a main method) that produce some deterministic output via System.out. These tests compare the output of these programs when run under Maxine VM and another trustworthy JVM (such as HotSpot).

Cross-ISA tests

When porting Maxine VM's compilers to a new Instruction Set Architecture QEMU is utilized to virtually run unit tests and regress the correctness of the generated code. To be able to run cross-ISA tests Maxine VM relies on gcc cross-compilers, gdb-multilib and qemu. The easiest way to run the cross-ISA tests is by using the `beehivelab/maxine-dev` docker image.

If you insist on running natively, assuming an Ubuntu 18.04 LTS installation, the following will install the required packages.

ARMv7 and AArch64

```
sudo apt-get install \
    gdb-multilib qemu-system-arm \
    gcc-aarch64-linux-gnu gcc-arm-none-eabi
```

RISC-V

GCC:

```
sudo apt-get install gcc-riscv64-linux-gnu
```

GDB:

```
sudo apt-get install wget texinfo
mkdir /tmp/riscv
cd /tmp/riscv/
wget https://ftp.gnu.org/gnu/gdb/gdb-8.2.1.tar.xz
tar xf gdb-8.2.1.tar.xz
cd /tmp/riscv/gdb-8.2.1
./configure --target=riscv32-elf,riscv64-elf --disable-multilib --prefix=/
↪opt/riscv
```

(continues on next page)

(continued from previous page)

```
make -j
sudo make install
```

QEMU:

```
sudo apt-get install libglib2.0-dev libpixman-1-dev flex bison
wget https://download.qemu.org/qemu-3.1.0.tar.xz
tar xf qemu-3.1.0.tar.xz
cd /tmp/riscv/qemu-3.1.0
./configure --target-list=riscv64-softmmu,riscv32-softmmu,riscv64-linux-
↪user,riscv32-linux-user --prefix=/opt/riscv
make -j
sudo make install
```

Don't forget to add `/opt/riscv` to `PATH`.

7.6.3 Logging and Tracing

Maxine provides two related mechanisms for logging and/or tracing the behavior of the VM, manual string-based logging using the `com.sun.max.vm.Log` class, or more automated, type-based logging, that is integrated with the *Inspector*, using `com.sun.max.vm.log.VMLogger`. These are related in that `VMLogger` includes string based logging as an option and so can replace the use of `Log`. Currently the VM uses a mixture of these two mechanisms, with conversion being done opportunistically. For simplicity, we will use the term tracing to describe string-based logging in the following. If you are adding logging to a VM component you are strongly encouraged to use the `VMLogger` approach.

Manual Tracing

Use the class `com.sun.max.vm.Log` to do manual tracing. The class includes a variety of methods for printing objects of various types. By default the output goes to the standard output but can be redirected to a file by setting the environment variable `MAXINE_LOG_FILE` before running the VM. To selectively enable specific tracing in the VM, define a `com.sun.max.vm.VMOption` with the name `-XX:+TraceXXX`, where `XXX` identifies the tracing.

You should avoid string concatenation (or any other code involving allocation) in tracing code, especially inside a `VmOperation`. While this should not break the VM (allocation will fail fast with an error message if a VM operation does not allow it), allocation can add noise to your logs. Lastly, if the logging sequence involves more than one logging statement, you should bound the sequence with this pattern:

```
boolean lockDisabledSafepoints = Log.lock();
// multiple calls to Log.print...() methods
Log.unlock(lockDisabledSafepoints);
```

This will serialize logging performed by multiple threads. Of course, it will also serialize the execution of the VM and may well make the race you are trying to debug disappear!

Native Code Tracing

Maxine provides some tracing of the small amount of native code that supports the VM. By default this is conditionally compiled out of the VM image but can be selectively enabled by editing `com.oracle.`

`max.vm.native/share/log.h` and rebuilding with `mx build` and rebuilding the VM image. This is particularly useful if the the VM crashes during startup. For example to enable all tracing set `log-ALL` to 1.

Type-based Logging

In type-based logging, the actual values that you want to log are passed to an instance of the `com.sun.max.vm.log.VMLogger` class using methods defined in the class. Evidently, at the `VMLogger` level, type-based logging is something of a misnomer, as it cannot know the types of the actual values. In practice the values are logged as untyped `Word` values, but extensive automated support is provided to handle the conversion to/from `Word` types. The optional tracing support is driven from the values in the log. For more details see *Type-based Logging*.

7.6.4 Debugging Maxine Java Tasks

The Maxine project includes a number of Java programs that can be launched as commands of the `mx` script. For example, the `mx image` command described above runs the `com.sun.max.vm.prototype.BootImageGenerator` class on a host JVM. This simplest way to debug such a command is to use the `-d` global option of the mx script. This will launch the Java program with extra options telling it to wait and listen for a JDWP-capable debugger on port 8000. You then configure a JDWP-enabled debugger to attach to this port.

The advantage of this approach is that you can easily launch the command with different command line arguments without having to create/modify an IDE launch configuration.

7.6.5 Core dump

To get a core dump from a Maxine VM process, it is simplest to do `gcore <pid>` from another shell. This forces a core dump but does not terminate the process, which continues after the dump is taken. An alternative is to use `kill -s ABRT <pid>` which does kill the process after the dump is taken. One other difference is that `gcore` allows the path to the core dump file to be specified with the `-c <corefile>` option, whereas `kill` puts it in a default location, typically `core` in the current working directory.

It is possible to force a core dump on a fatal VM error by setting the option `-XX:+CoreOnError` when running the VM.

The following invocation:

```
mx inspect --mode=attach --target=file --location=dumpfile
```

will then bring up the Inspector on the core dump. If you omit the `--location` argument, it will put up a dialog box.

Unfortunately this will only work if the associated Maxine VM was run with the `-XX:+MakeInspectable` option, otherwise some key data structures needed by the Inspector will not have been created.

7.7 Frequently Asked Questions

Here are some of the most frequently asked questions, along with answers that have been updated as the project evolves. You might also want to browse the Maxine "*Glossary*".

7.7.1 Does Maxine support the GNU classpath libraries?

No. Maxine is designed to run with openJDK.

7.7.2 How modular is the Maxine VM architecture?

Maxine provides abstractions for coarse-grained configurability of many VM subsystems, which we call schemes. For example, the static and runtime compilers, garbage collector, reference representation, object layout, monitor implementation, are all configurable with schemes.

7.7.3 What kind of GC does the Maxine VM use?

Currently Maxine employs a semi-space collector as the default. As with many other parts of the Maxine architecture, garbage collection is abstracted as a separate scheme with limited interactions with other schemes. We also aim to make the garbage collector scheme MMTK compatible.

7.7.4 How much optimization does the baseline compiler do?

Essentially none. See *T1X compiler*.

7.7.5 Does the Maxine VM use Green Threads?

No; Maxine does not use green threads. Maxine uses native threads and a state-of-the-art safepoint mechanism for preemption. See *Threads*.

7.7.6 Can I use my favorite debugger?

No, instead we use the The Maxine Inspector for debugging and inspecting the VM while it is running.

7.7.7 What kind of development environment do I need to build and run the Maxine VM?

Maxine can be built and run from the command line, no special development environment is needed.

7.7.8 How does Maxine relate to the Project Maxwell Assembler System?

The Maxine distribution provides an advanced variant of the Project Maxwell Assembler System, now called the Maxine Assembler System.

7.7.9 Are there other attempts to bootstrap the Java programming language?

Yes, some examples include Jikes RVM, Joeq, OVM, and Moxie. The design of Maxine has benefited from these previous systems and enjoys the advantage of the new source language features in Java 5.0, particularly annotations and generics.

7.7.10 Can I suspend and resume VM execution?

The simplest way to get application suspend/resume currently is to run Guest VM (Maxine on Xen). Suspend/resume is built into the hypervisor support and "just works". The Guest VM is server-side only though, no GUI at this point.

7.7.11 Does the Maxine VM have an interpreter?

Maxine doesn't do interpretation. It instead uses a very fast baseline compiler. See *T1X compiler*.

7.7.12 Why am I getting an error message about "hosted" being missing when trying to build an image?

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: no hosted in␣
↪java.library.path
```

The native library named "hosted" is used during the boot image generation to get information on the host platform. Very likely this has not been built for some reason.

First, be sure that you have a C development environment installed. Also ensure that the CDT plugin is installed if you are using Eclipse.

Lastly, try rebuilding the native code:

```
mx build -clean com.oracle.max.vm.native
```

7.7.13 Why was the Grip abstraction from the original VM design removed?

Re: changeset 4423

The original rational for Grips was to provide an abstraction for object references that does not involve write barriers. Apart from that, it was pretty much an exact mirror of the `ReferenceScheme`. The thinking was that the GC should be written against grips as it does not need to update write barriers. However, it turns out that object reference fix up is done via Pointers in the current GC implementations and I don't see why this won't/can't be true for any other GC. That is, we had a whole extra (and confusing) abstraction whose whole reason for existence was not being used! Additionally, even if references were being fixed up via `Reference.setReference(...)` and `Reference.writeReference(...)` instead of `Pointer.setReference(...)` and `Pointer.writeReference(...)` there is still no need for an extra abstraction. It would be far simpler to annotate the method(s) doing the update with an annotation (e.g. `@NO_BARRIERS`) that would instruct the compiler not to insert write barriers.

Of course, Maxine's abstractions should support more than just write barriers for generational GCs. Other interesting barriers include read barriers for concurrent GCs, read & write barriers for all data

types in an software transactional memory implementation, etc. I cannot say for certain that the support for these is sufficient right now, but I'm confident they can be programmed without grips.

7.7.14 How does the Inspector process communicate with the inspected Maxine VM process?

The VM is almost entirely passive with respect to the Inspector process. There is no internal agent; the VM neither sends nor receives messages; in fact the VM barely knows that it is being inspected. Other than process controls (thread management, start, stop, set breakpoints, etc.), the Inspector works mostly by reading from VM memory. However, VM code is arranged in some places to make inspection easier, and there are a few critical places where the VM does respond to information written into its memory by the Inspector. See Inspector-VM Interaction.

7.7.15 What happened to the "primordial thread"?

Until February 2011 the original thread in a new Maxine VM process was known as the primordial thread; its job was to execute the preliminary steps needed to bootstrap the VM and then wait until the Java VM exited. From February 2011 onward, the original process thread eventually becomes the main thread, i.e. the thread on which the Java main thread runs. See *Threads*.

7.8 Glossary of Maxine terminology and concepts

Welcome to the Maxine Glossary, a very informal and evolving set of brief notes to help orient newcomers to some of the terminology and concepts we use when talking about the Maxine VM. Please feel free to write to us with comments and suggestions. It is definitely a work in progress.

You might also want to browse the *Maxine FAQ*.

We thank our collaborators who have been contributing documentation as well; we link to it from this page and others whenever possible.

7.8.1 Alias

A specially marked field or method in a VM class that refers to a field or method in another class that would otherwise be inaccessible due to Java language access control rules. Used mainly for VM access to private members of JDK classes.

Read more.

7.8.2 Annotations

The Maxine VM code makes heavy use of Java Annotations as a form of language extension. These extensions, which are recognized and treated specially by the Maxine compilers, permit the kind of low-level, unsafe programming that is otherwise not possible with Java. By using the Java annotation mechanism, which is a first class part of the language, the Maxine sources are completely compatible with Java IDEs. See package `com.sun.max.annotate`.

Here are a few of the important Maxine annotations:

- `@ALIAS`: denotes a field or method as an alias, which can be used to access a field or method in another class that would otherwise be inaccessible due to Java language access control rules.
- `@BUILTIN`: denotes a method whose calls are translated directly by the compiler into machine code.
- `@C_FUNCTION`: denotes a native function for which a lightweight JNI stub should be generated.
- `@CONSTANT_WHEN_NOT_ZERO`: denotes a field whose value is final once it is non-zero.
- `@CONSTANT`: denotes a field whose value is final before it's first read (i.e. a stationary field).
- `@FOLD`: calls to these methods are evaluated (as opposed to translated) at compile time.
- `@INLINE`: forced inlining.
- `@INSPECTED`: used by an offline tool to generate field and method accessors for the Maxine Inspector.
- `@METHOD_SUBSTITUTIONS`: denotes a class containing. `MethodSubstitutions`
- `@NEVER_INLINE`: denotes a method that this compiler must never inline.
- `@SUBSTITUTE`: denotes a `MethodSubstitution`.
- `@UNSAFE`: marks a method that requires special compilation; some other annotations imply `@UNSAFE`.

7.8.3 Boot heap

An object heap embedded in the VM boot image. It is a normal heap, with the exception that objects in it never move (although they may become permanent garbage). As the name implies, the objects in this heap are those allocated during boot image generation. The boot image is really just this heap plus a little meta-data in front.

Read more.

7.8.4 Bootstrap

The process of loading and executing a boot image of Maxine, up to the point where the VM is ready, either to execute a specified application class or other action specified by the run scheme.

Currently a boot image of Maxine is not a native executable but just a binary blob containing machine code and data for a dedicated target platform. Thus a boot image is not executable by itself. To start it a very small native C application is required.

See *Boot Image*.

7.8.5 Graal Compiler

See Graal.

7.8.6 Immortal memory

See the `ImmortalHeap` class as well as the various `ImmortalHeap_*` classes that test this functionality.

See class `com.sun.max.vm.heap.ImmortalHeap`

7.8.7 Injected fields

During startup the VM synthesizes and injects additional fields into core JDK classes. Injected fields typically link instances of JDK objects to their internal VM representation. *Read more*.

7.8.8 Maxine packages

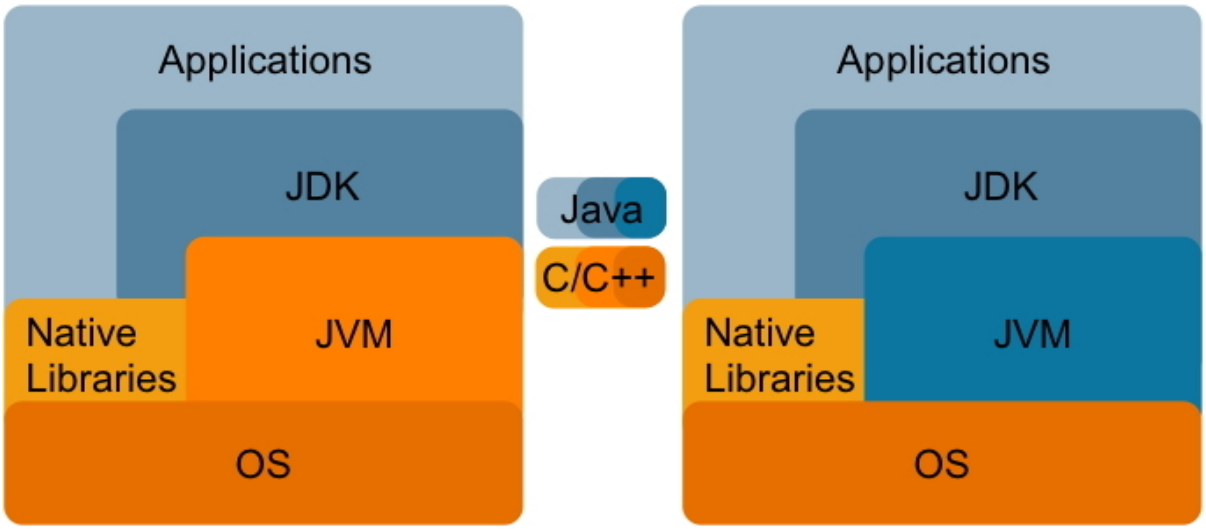
A mechanism for treating groups of classes in Java package as a de facto "module" for purposes of system configuration and evolution. This requires implementing more functionality than is provided by the Java language via `java.lang.Package`.

This main application of this mechanism is to define the classes to be including during Maxine *boot image generation*, and in particular to specify which implementations to bind to VM schemes.

Strictly speaking, a Maxine package is a collection of classes in a Java package that includes a class named `Package`. The class Package must extend class `com.sun.max.config.BootImagePackage` in order to be considered for inclusion in the VM. The `Package` class, other than acting as a marker, may contain additional specifications directed at the Maxine package system. In many cases, however, trivial `Package` class can be synthesized dynamically and need not be explicitly defined.

7.8.9 Metacircular VM

In a conventional VM implementation (left in the figure below) there is a language barrier between the language being implemented (Java in the figure) and the implementation language (C++). No such barrier exists in Maxine, where the VM is itself implemented in the language being implemented.



See also: Ungar, D., Spitz, A., and Ausch, A. 2005. Constructing a metacircular Virtual machine in an exploratory programming environment. In *Companion To the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA, October 16 - 20, 2005). OOPSLA '05. ACM, New York, NY, 11-20. DOI

7.8.10 package-info.java

A documentation class, following Javadoc convention, for the classes and interfaces in a Java package; this is especially encouraged for packages that constitute Maxine package and serve as modules for VM configuration.

7.8.11 Package.java

A class used for configuration purposes by the Maxine Package mechanism.

7.8.12 ReferenceMapInterpreter

The ReferenceMapInterpreter performs an iterative data flow analysis via abstract interpretation. The following option maybe useful to watch it in action:

```
-XX:TraceRefMapInterpretationOf=<value>
```

The help message for this option is: "Trace ref map interpretation of methods whose name or declaring class contains ."

A short summary of its operation follows, contributed by Arian Treffer.

- To collect GC roots, the GC needs to know which variable and stack slots in a stack frame contain references.
- For the beginning of each code block, a bitmap (called "frame") that indicates used reference slots is cached.
- A block is a sequence of byte codes in a method that can be executed without jumping (out or into). A block either ends with a (implicit) fall through, a jump, or a return.
- To create frames, the blocks are pseudo interpreted: their pop and push behavior is simulated. The slot configuration at the end of a block is the frame for all blocks that can be reached from here (2 in case of a conditional jump, 0 in case of a return, otherwise 1).
- When a block can be reached from multiple other blocks, its frame is the intersection of the final slot configuration of its predecessors. If one predecessor stored a reference in a slot, and another did not, the current block may not read this slot, for it doesn't know its contents.
- The stack size at the beginning of a block is always the same. There is no Java code that first pushes N items (i.e. in a loop), and later pops them, even though this could be expressed with byte codes.
- To get the slot configuration at the current execution point, the current block is interpreted again up until the current byte code, where the slot configuration is converted into a bitmap that indicates references on the current stack frame.

7.8.13 Stop positions

A list of call and safepoint instructions within a target method. These locations correspond to all possible addresses the instruction pointer of a frame may have when its thread is stopped at a safepoint. The location of all references on the stack are precisely known when at a stop position. See *Threads*.

7.8.14 T1X compiler

T1X is a template-based baseline compiler and is Maxine's first line of execution (Maxine has no interpreter). As such, it's primary goal is to produce code as fast as possible. Code quality is of secondary concern. It also closely matches the JVM specification's execution models. That is, the JVM operand stack and local variable variable array is modeled directly. This makes it suitable for implementing bytecode level debugging as well being the execution mode the de-optimization process uses as its end target.

The templates for each bytecode instruction are written in Java (see `T1XTemplateSource`) and compiled to machine code by C1X (which is to be replaced by Graal). These machine code snippets are stored in a table and used to translate bytecodes at T1X compile time. The translation is done in a single pass (see `T1XCompilation`) and GC maps are lazily generated via an abstract interpreter at GC time. The latter strategy pays off as a GC map is only generated for a T1X compiled method if it is currently active during GC root scanning. Another strategy to improve compile time is to minimize allocation during compilation. This is achieved by (re)using thread local data structures for each compilation.

Having the templates written in Java makes modifying or extending the compiler fairly easy. More importantly, it also means the compiler is very portable and it mostly relies on the optimizing compiler. It performs very little direct machine code generation.

The source code for T1X is entirely contained in the top level T1X directory of the Maxine source code base.

7.8.15 Target method

A target method in the Maxine VM is the entity that contains some machine code produced by one of the compilers in Maxine. It also contains all the other data required by the VM for some machine code. In particular, target methods (implemented by heap objects in the class hierarchy rooted at `TargetMethod.java`) encapsulate the following information, including some that resides not in the heap but in the region of code cache memory allocated for the compilation.

- Machine code, represented as a reference to a `byte[]` that is stored in the method's code cache allocation.
- Reference literals (optional, but common): represented as a reference to an `Object[]` that is stored in the method's code cache allocation.
- Scalar literals (optional, much less common): represented as a reference to a `byte[]` that is stored in the method's code cache allocation.
- Exception handler information. This is a data structure that can be used to answer the question "for an exception of type t thrown at position n in the target method, what is the position, if any, of an exception handler in the target method that will handle the thrown exception?".
- The stop positions. A stop is a machine code position for which extra information is known about the execution state at that position. There types of stop positions in Maxine and the information recorded for them are shown below:

- **Call**. This is the position of a call (direct or indirect) instruction. For a call, the following is recorded:
  - Frame reference map. This is a bit map with one bit per slot in the frame of the method. A set bit in this bit map indicates that the corresponding frame slot holds an object reference at the call.
  - Java frame descriptor. This is a map from locations in the bytecode-level frame state to locations in the machine state. The bytecode level frame state is composed of the local variables and operand stack slots addressed by the JVM bytecodes from which the machine code was produced. The machine state is composed of frame slots, registers and immediate instruction operands. The mapping enables the JVM state to be completely reconstructed at the stop position. This is useful for implementing source level debugging and deoptimization.
- **Safepoint**. This is the position of a safepoint instruction. For a safepoint, all the information described for a call is recorded as well as:
  - Register reference map. This is a bit map with one bit per register that can be used to store an object reference. This includes the complete set of general purpose registers for the platform but exclude all the floating point and state registers. Like a frame reference map, a set bit in the register reference map indicates that the corresponding register is holding an object reference at the safepoint.

[STRIKEOUT:Currently register reference maps are not recorded for calls as all registers are caller saved by the compilers in Maxine. This will mostly likely change in the near future as C1X will implement callee-save registers when compiling certain methods.] (Out of date?)

See abstract `com.sun.max.vm.compiler.target.TargetMethod`

7.8.16 Trampoline

The mechanism used to defer binding a call site to a target method. When compiling a call, an address is needed for the machine level call instruction. One option is to eagerly resolve the callee during compilation of the call but this will end up compiling the world! Instead, a piece of code is called that knows how to find and compile (if necessary) the intended target method and redirect the call there. For static calls, the call site itself is patched so that subsequent calls go straight to the resolved method. For virtual calls, the trampoline patches the entry in the relevant dispatch table.

7.9 Actors

An actor is an object that represents a Java language entity (e.g. a Class, Method, or Field) in the VM and implements the entity's runtime behavior. All Maxine actors are instances of classes that extend abstract class `com.sun.max.vm.actor.Actor`.

Maxine actors can be viewed as enhanced reflection classes (i.e. classes such as `java.lang.reflect.Method` and `java.lang.Class`). Java reflection classes by design hide implementation details specific to any VM (including in most cases information about the underlying class file). Maxine actors, on the other hand, exist precisely to implement those internal details specifically for the Maxine VM.

7.9.1 Actors and their JDK counterparts

The implementation of Maxine actors and their JDK `java.lang.reflect` counterparts are typically intertwined. Since the Maxine VM is designed to operate with a standard, unmodified JDK, modifications must be made dynamically to some JDK classes at VM startup so that the two can be coordinated.

Three techniques make this possible:

1. **Aliases:** direct, non-reflective access to JDK fields and methods, even when prohibited by standard Java access rules;
2. **Field injection:** adding a field to a JDK class, for example a pointer from an instance of `java.lang.reflect.Class` to its corresponding `ClassActor`; and
3. **Method substitution:** replacement of a JDK method.

See *JDK interoperation* for details and examples.

7.9.2 Flags

The abstract class `Actor` contains exactly one field, a word used as a bit field, along with a number of accessor methods for those values. These provide efficient and flexible access to properties of interest for all actors.

Many of the flags correspond to properties defined by the Java language, for example the presence of keywords such as public, private, and final. These are documented at the head of the the file and are cross-referenced to the *Java Language Specification*. Other flags are used strictly for internal implementation.

7.9.3 The Actor types

This section mentions a few members of the `Actor` type hierarchy; the actual type hierarchy is a bit more complex.

ClassActor

A `ClassActor` represents many of the implementation details for a Java class. For example, it includes a reference to the corresponding instance of `java.lang.Class`, which in turn contains an injected field reference that points back at the `ClassActor`.

A `ClassActor` also holds references to the class's methods (instances of `MethodActor`), fields(instances of `FieldActor`), its superclass, its static and dynamic hubs, and more.

`ClassActor` is abstract, with three subclasses:

1, `InterfaceActor` represents Java interfaces.

1. `PrimitiveClassActor` represents primitive Java types, as described by `KindEnum` (corresponding to the primitive Java types plus some created only for VM internal use).
2. `ReferenceClassActor` represents non-primitive Java types using three concrete subclasses:
    1. `ArrayClassActor<Value_Type>` represents Java arrays;
    2. `TupleClassActor` represents ordinary Java objects;

3. `HybridClassActor` represents a kind of object that cannot be expressed in Java: a combination of array plus fields that is used internally to represents Maxine hubs.

An ordinary object instance in the VM's heap contains a header that, among other things, identifies the object's type. This field points not at the `ClassActor` for the object's type, but rather at the dynamic hub for the class. In the case of the exceptional object that holds the static fields of a class (the static tuple), the header points to the static hub for the class.

FieldActor

A `FieldActor` contains the implementation details for a field in a Java class. Such details include a reference to the representation of the field's type and to its holder: the instance of `ClassActor` representing the implementation of the class to which the field belongs.

A subclass of `FieldActor`, `InjectedReferenceFieldActor`, represents a synthesized field that has been added dynamically to a JDK class.

See *field injection* for details and examples.

MethodActor

A `MethodActor` contains the implementation details for a method in a Java class. Such details include a reference to the representation of the method's signature, to it's holder (the instance of `ClassActor` representing the implementation of the class to which the method belongs), and to zero or more possible compilations of the method.

The `MethodActor` class is itself abstract, with concrete subclasses defined to implement various flavors of implementation: static methods, virtual methods, interface methods, and so-called *miranda methods*.

7.10 JDK interoperation

The Maxine VM is designed to work with a standard, unmodified JDK, which requires special machinery for dealing dynamically with important JDK classes. This machinery is implemented by compiler extensions, configured by *annotations*.

JDK classes in the JDK can be modified during VM bootstrapping, both by adding fields and by replacing methods.

7.10.1 Aliases

Aliases give the VM access to otherwise inaccessible fields and methods. Java's access rules for class members can be bypassed using reflection, but at the cost of boxing/unboxing and the absence of static type checking. The Maxine VM avoids these costs with the annotation `@ALIAS`. This causes an annotated field or method to act as an alias for a field or method in another class, giving straightforward, statically type-checked access.

For example, the `name` field in class `com.sun.max.vm.jdk.JDK_java_lang_Thread` provides read/write access to the private field `name` in class `java.lang.Thread`.

```
@ALIAS(declaringClass = java.lang.Thread.class)
char[] name;
```

In case the `declaringClass` is private `declaringClassName` can be used instead.

```
@ALIAS(declaringClassName = "java.lang.reflect.Proxy$ProxyClassFactory")
public final static String proxyClassNamePrefix = "$Proxy";
```

Additionally in case the type of the object itself is private, the `descriptor` parameter can be used.

```
@ALIAS(declaringClass = Proxy.class, descriptor="Ljava/lang/reflect/
↪WeakCache;")
private static Object proxyClassCache = null;
```

Inner classes are also supported through the `innerClass` parameter.

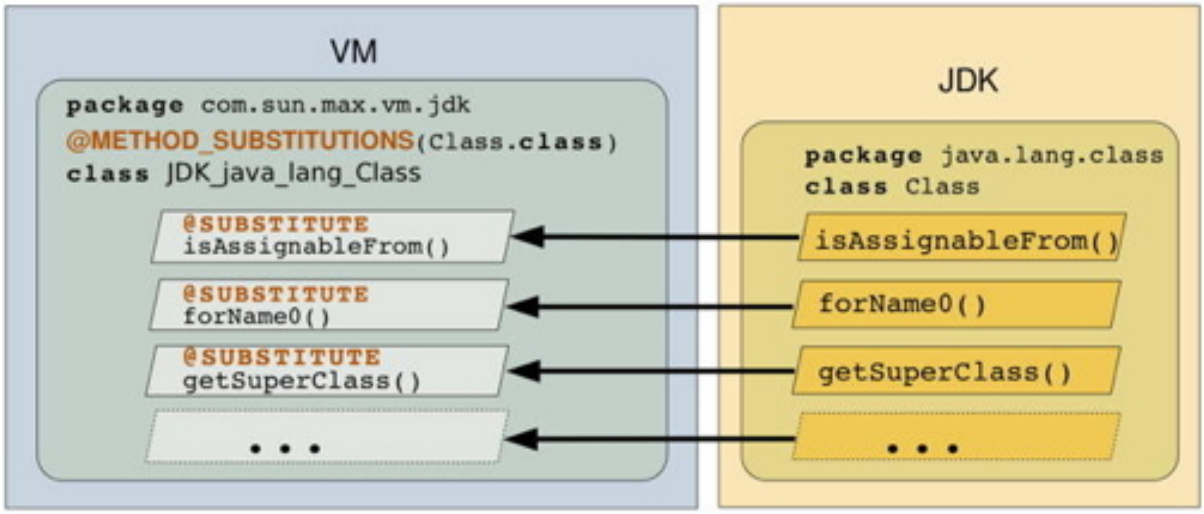
```
@ALIAS(declaringClass = ClassLoader.class, innerClass = "NativeLibrary")
private long handle;
```

For more discussion and an example of method aliasing, see the Javadoc comments for the annotation `com.sun.max.annotate.@ALIAS`.

Aliasing is often used together with method substitution, so that a method substituted into a different class will have access to otherwise inaccessible members of that different class. See below an example of how this is done.

7.10.2 Method substitution

During startup the Maxine VM replaces specified methods in specified JDK classes. How this is configured is best described by the following example, in which the method `java.lang.Class.isAssignableFrom()` gets replaced (along with others). The following figure depicts the result of the substitution.



1. Create a Maxine VM class for holding all methods to be substituted into `java.lang.Class`. By convention, this class is named `JDK_java_lang_Class` (in package `com.sun.max.vm.jdk`).
2. Annotate this class with `@METHOD_SUBSTITUTIONS`, which identifies

the target of the substitutions as follows:

```
@METHOD_SUBSTITUTIONS(java.lang.Class.class)
```

3. Create a method in class `JDK_java_lang_Class` with signature identical to the method to be substituted, in this case `isAssignableFrom()`. The body of this method does what is actually needed during execution of the Maxine VM. This often entails delegating part of all of the operation to a VM object that is the Maxine runtime implementation of the JDK object, in this case an instance class `com.sun.max.vm.actor.holder.ClassActor`.
4. Annotate the newly created method with `@SUBSTITUTE`: which marks the method for substitution.

Substituted methods don't necessarily have access to fields and methods of the class into which they are substituted, but language access rules can be defeated by creating aliases for those fields and methods as needed. For example, the method `setName()` in class `com.sun.max.vm.jdk.JDK_java_lang_Thread` assigns the name of a thread, both in the VM's representation (a `VmThread`) and in the JDK's instance of class `java.lang.Thread` via assignment to a field alias.

```
@ALIAS(declaringClass = Thread.class)
char[] name;

...

/**
 * Sets the name of the the thread, also updating the name in the␣
↪corresponding VmThread.
 * @param name new name for thread
 */
@SUBSTITUTE
private void setName(String name) {
    thisThread().checkAccess();
    if (thisVMThread() != null) {
        thisVMThread().setName(name);  // Set name in the VM's thread␣
↪object
    }
    this.name = name.toCharArray();    // Set name in the JDK's thread␣
↪object
}
```

Note that when substituting a constructor, the new constructor no longer invokes the original initializers (if any) of the corresponding class, so one needs to do this explicitly.

```
@SUBSTITUTE(constructor = true, signatureDescriptor = "(Ljava/lang/invoke/
↪LambdaForm;ILjava/lang/String;Ljava/lang/String;Ljava/lang/invoke/
↪MethodType;)V")
private void InvokerBytecodeGenerator(Object lambdaForm, int localsMapSize,
                                      String className, String invokerName,
↪ MethodType invokerType) {
    if (invokerName.contains(".")) {
        int p = invokerName.indexOf(".");
        className = invokerName.substring(0, p);
        invokerName = invokerName.substring(p + 1);
    }
    className = maxMakeDumpableClassName(className);
    this.className = superName + "$" + className;
```

(continues on next page)

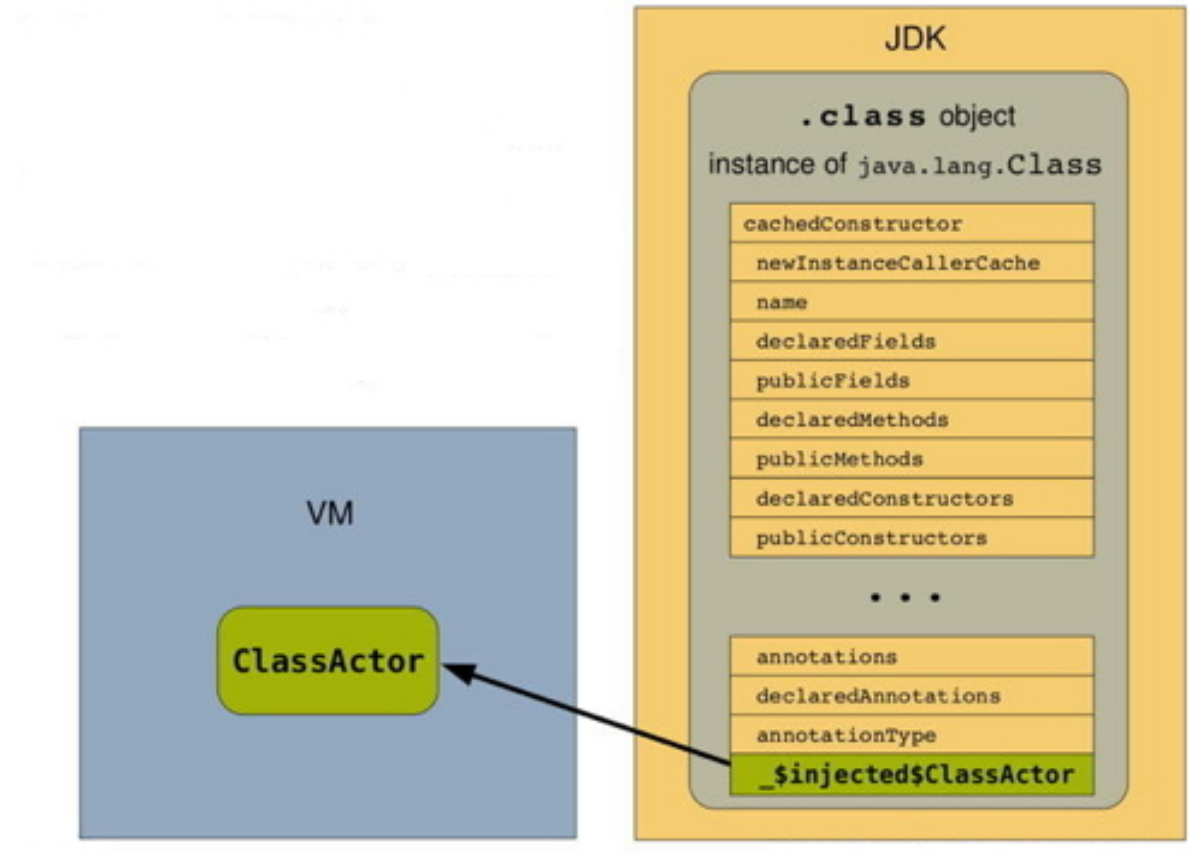
(continued from previous page)

```
    this.sourceFile = "LambdaForm$" + className;
    this.lambdaForm = lambdaForm;
    this.invokerName = invokerName;
    this.invokerType = invokerType;
    this.localsMap = new int[localsMapSize];

    // When substituting a constructor the initializers of the original␣
↪class are no longer invoked, thus we need
    // to initialize cpPatches explicitly here
    cpPatches = new HashMap<>();
}
```

7.10.3 Field injection

During startup the VM synthesizes and injects additional fields into core JDK classes. Injected fields typically link instances of JDK objects to their internal VM representation. For example, the VM injects into class `java.lang.Class` a reference to each class's internal VM representation: an instance of class `com.sun.max.vm.actor.holder.ClassActor`. The following figure depicts the result of the injection.



A field injection is defined by creating an instance of class `com.sun.max.vm.actor.member.InjectedFieldActor`.

For example, the following code (which creates a static variable in class `com.sun.max.vm.actor.member.InjectedFieldActor`), causes the VM to inject the `ClassActor` reference field into class `java.lang.Class`, as shown in the above figure.

```
/**
 * A field of type {@link ClassActor} injected into {@link Class}.
 */
public static final InjectedReferenceFieldActor<ClassActor> Class_
↪classActor = new InjectedReferenceFieldActor<ClassActor>(Class.class,␣
↪ClassActor.class) {
    @HOSTED_ONLY
    @Override
    public ReferenceValue readInjectedValue(Object object) {
        final Class javaClass = (Class) object;
        return ReferenceValue.from(ClassActor.fromJava(javaClass));
    }
};
```

7.11 VM Boot Image

Starting an instance of the Maxine VM requires the use of a previously constructed boot image, a near-executable binary VM image that includes an initially populated heap, together with compiled code for a dedicated target platform.

7.11.1 Boot image contents

The boot image is a binary file that contains a near-executable memory image of the running VM in the format of the target platform for which the image was generated. The target platform is not required to be the same as the platform on which the boot image generation is run (i.e. hosted).

The utility class `BootImage` is responsible for writing the boot image file in the format of the target platform. This data must be read by native C code during VM startup, so there is necessary agreement between the layout as expressed in the Java class BootImage and in the native startup code.

Below you can see the boot image layout summarized as a pseudo-C struct. This diagram is excerpted from the Javadoc comments for class `BootImage`.

```
struct BootImage {
     struct Header header;            // see declaration of image_Header␣
↪in Native/substrate/image.h
     struct StringInfo string_info;   // see declaration of image_
↪StringInfo in Native/substrate/image.h
     byte[header.relocationDataSize] relocation_data;
     byte[] pad;                      // padding such that next field will␣
↪be aligned on an OS page-size address
     byte[header.heapSize];           // header.heapSize is a multiple of␣
↪page-size
     byte[header.codeSize];           // header.codeSize is a multiple of␣
↪page-size
     struct Trailer trailer;          // see declaration of image_Trailer␣
↪in Native/substrate/image.h
}
```

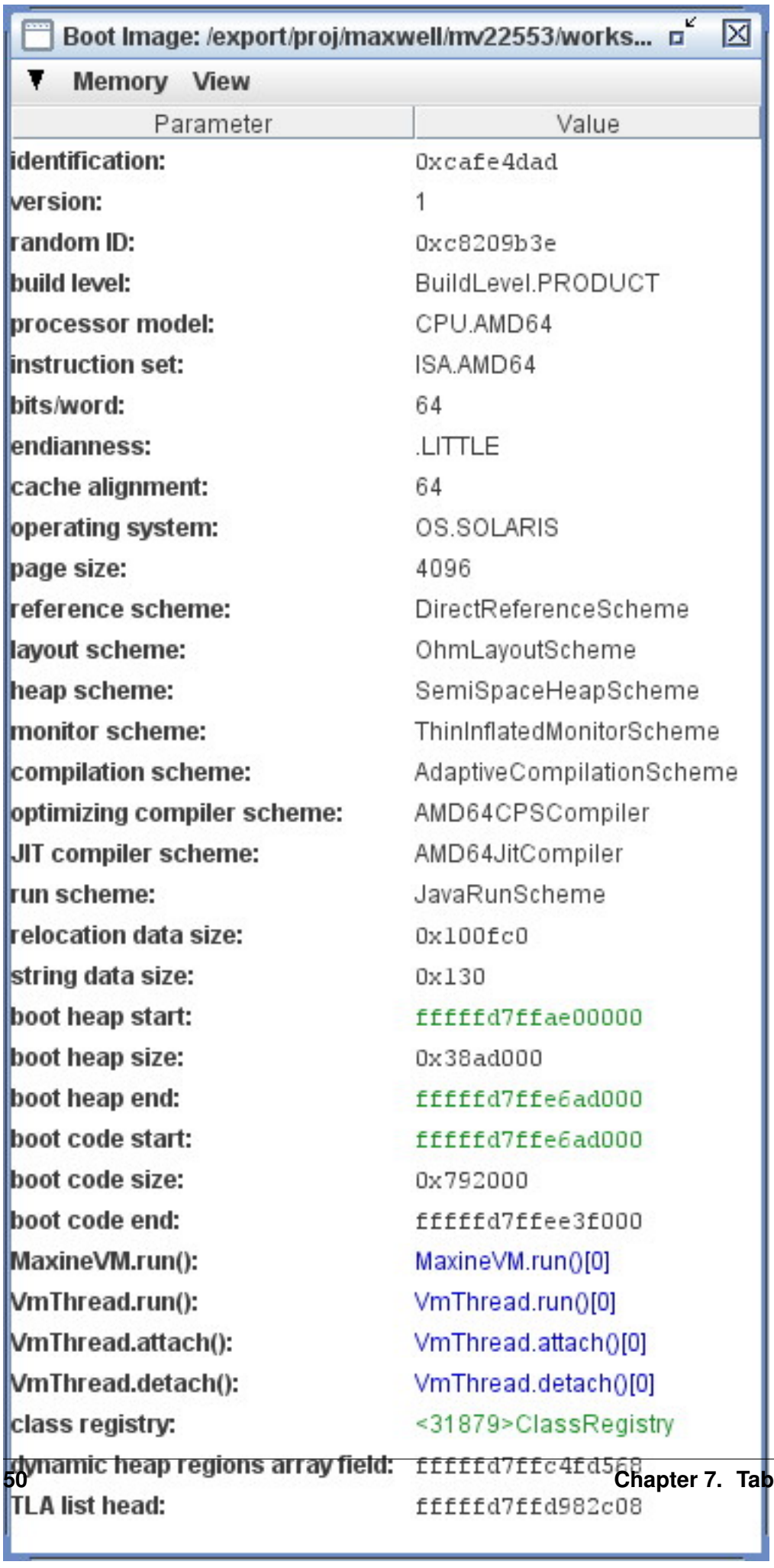
The boot image contains the following groups of information:

- **header:** a sequence of integer parameters that identify the binary, describe the target platform, and include specific data about the remaining contents of the image.

- **string info:** a sequence of strings in native format that describe the build, name the classes that provide detailed platform description, and name the packages that are bound to the schemes in the build.
- **relocation data:** a bit field that identifies address locations in the image that need to be relocated.
- **boot heap:** binary image of a heap in target platform format, pre-populated with the objects that represent class and other runtime data needed by the VM until it can begin loading classes on its own.
- **boot code:** binary image of method compilations in target platform format, pre-populated with the compilations needed by the VM until it can begin compiling methods on its own.
- **trailer:** a subset of the integer parameters that also appear in the header, repeated for verification.

7.11.2 Inspecting the boot image

The Boot Image Inspector view in the Maxine Inspector displays summary information about the specific boot image being viewed: identifying parameters, configuration classes for various schemes, memory region information about the boot heap and boot code, and other specific references into the heap and compiled code.


Boot Image: /export/proj/maxwell/mv22553/works...
Memory View
Parameter
Value
identification: 0xcafe4dad
version: 1
random ID: 0xc8209b3e
build level: BuildLevel.PRODUCT
processor model: CPU.AMD64
instruction set: ISA.AMD64
bits/word: 64
endianness: .LITTLE
cache alignment: 64
operating system: OS.SOLARIS
page size: 4096
reference scheme: DirectReferenceScheme
layout scheme: OhmLayoutScheme
heap scheme: SemiSpaceHeapScheme
monitor scheme: ThinInflatedMonitorScheme
compilation scheme: AdaptiveCompilationScheme
optimizing compiler scheme: AMD64CPSCompiler
JIT compiler scheme: AMD64JitCompiler
run scheme: JavaRunScheme
relocation data size: 0x100fc0
string data size: 0x130
boot heap start: ffffffd7ffae00000
boot heap size: 0x38ad000
boot heap end: ffffffd7ffe6ad000
boot code start: ffffffd7ffe6ad000
boot code size: 0x792000
boot code end: ffffffd7ffee3f000
MaxineVM.run(): MaxineVM.run()[0]
VmThread.run(): VmThread.run()[0]
VmThread.attach(): VmThread.attach()[0]
VmThread.detach(): VmThread.detach()[0]
class registry: <31879>ClassRegistry
dynamic heap regions array field: ffffffd7ffc4fd568
TLA list head: ffffffd7ffd982c08

7.11.3 BOOT image generation

Maxine's boot image generator creates a boot image according to a specification (see `com.sun.max.vm.VMConfiguration`) that both specifies which implementations of particular Schemes should be included and describes the platform on which the image is intended to run.

Class loading and initialization

Any class that might be written into the boot image must first be loaded by the boot image generator. Loading can happen in two ways. First any class that is used by the classes that comprise the boot image generator is loaded by the normal host class loading mechanisms, that is using the system class loader. Note that the boot image generator uses many of the same classes as the VM. Second, the generator explicitly loads classes that might be written into the boot image by scanning the class path and searching for sub-packages of com.sun.max.config that contains a class named Package that is a subclass of com.sun.max.config.BootImagePackage. The Package classes act as roots for the set of packages to be included in the boot image. BootImagePackage provides a mechanism for including packages that are outside the com.sun.max namespace. See below for details. This second set of classes is actually loaded by a special class loader, com.sun.max.vm.hosted.HostedVMClassLoader, that performs additional actions, such as creating the Maxine representation of classes used at runtime. HostedVMClassLoader delegates to the system or boot classloader, as appropriate, so a class is only loaded and initialized once, even if it was initially loaded implicitly by the boot image generator.

As noted, Loaded classes are initialized, which means that static class variables in the boot image have values assigned during image generation. Mostly this is perfectly fine, for example, an instance of a collection class. The problem arises with classes that capture values that are host-specific and inappropriate for the target VM environment. Typically this is any value related to the external environment, which includes the operating system and host virtual machine. An example of the latter would be any object that contains a VM-specific native value, for example a soft reference. The boot image generator takes care of all the known cases in the JDK classes, and this is handled in the class `com.sun.max.vm.hosted.JDKInterceptor`. Extension classes can register a class to have the class initialization re-run at run-time. Except for such classes, no code is generated in the boot image for static initialization.

Class and method selection

Intuitively, the boot image is intended to contain all the classes and methods that are needed to bootstrap the VM to the point where it can dynamically load further classes from the file system and compile methods for execution. In order to minimize the size of the boot image, only those classes should be included. In practice this set is not easy to define precisely and depends, amongst other things, on the details of the JDK implementation. In particular, it can vary from release to release of the JDK.

The Maxine boot image generator determines the content of the boot image by starting from so-called entry points plus a subset of the standard platform classes that are known to be necessary for the bootstrap. An entry point is a method that:

- can be called from the external environment (e.g. those annotated with `@VM_ENTRY_POINT`),
- is called by pre-compiled/generated code such as stubs,
- is a reflection invocation stub for methods called during class loading or compilation,
- is manually selected to satisfy bootstrap requirements (and avoid infinite recursion at runtime), or

- is manually selected to improve startup time at runtime.

These entry points form the basis of the analysis used to discover the methods needed to bootstrap the VM. An example is the run method in `com.sun.max.vm.MaxineVM`, which is the entry point from the native C code that starts the VM bootstrap. Another mechanism for specifying an entry point is the `com.sun.max.vm.runtime.CriticalMethod` class.

7.11.4 Extending the boot image

Maxine supports both static and dynamic extension of the VM boot image. Static extension adds extra classes to the boot image generation process. Dynamic extension is supported using a similar mechanism to that for Java agents,

7.11.5 Static Extension

Static extension provides the opportunity to customize the set of classes to be included in an image. For example, if you need to extend the VM in a way that requires some of the extension classes to execute before the VM is ready to load new classes, then those additional classes must be included in the boot image.

It is possible to augment the boot image with much larger set of classes, in the limit, every class needed by an application. Such an image would be self contained and require no dynamic class loading at runtime. In certain cases this might be desirable; however, there are issues regarding class initialization that may need to be addressed.

Class re-initialization

As noted above, static class variables in the boot image will have values assigned during boot image generation unless handled specially by the `com.sun.max.vm.hosted.JDKInterceptor` class. It is your responsibility to check any extension classes you are adding.

Specifying additional classes or packages

The simplest way to do this for a one-time image build is to suffix the class names or packages to the max image command:

```
% mx help image
mx image [options] classes|packages...

build a boot image

    Run the BootImageGenerator to build a Maxine boot image. The classes
    and packages specified on the command line will be included in the
    boot image in addition to those found by the Package.java mechanism.
    Package names are differentiated from class names by being prefixed
    with '^'.
...
```

For example, to add all the classes in the package `acme.demo` as well as the single class `foo.Bar`:

```
% mx image ^acme.demo foo.Bar
```

Note that packages are not processed recursively, that is, nested packages are not included unless explicitly specified. Note also that the above command assumes that the classes are located within one of the Maxine directories. If this is not the case, their location must be specified with the `--cp-sfx` option. In addition, methods in the extension classes will only be compiled if they meet the conditions outlined above in *Class and method selection*. Finally, protection issues may require your classes to be defined in a Maxine package if your extension classes need package-private access to any Maxine classes. For more permanent additions to the boot image, or to meet access requirements, the easiest approach is to put your classes in in a sub-package of `com.sun.max` in one of the existing Maxine projects, and leverage the automatic loading mechanisms described in the earlier section *Class loading and initialization*. While this approach might be acceptable for short-term prototyping, it is inappropriate in the long term unless the code is blessed as a standard VM component.

If you place your classes in a sub-package of com.sun.max, whether in an existing project or not, there is one caveat:

- If your classes are not in an existing Maxine directory, be sure to use the `--cp-sfx` option to add the directories to end of the classpath. If you add them at the front (with `--cp-pfx`), you may well break the mechanism by which the hosted native library is found.

The recommended approach for extending Maxine is to place the extension packages in a separate project. To satisfy the discovery mechanism of the boot image generator, all extension packages must be rooted in a sub-package of `com.sun.max.config`, for example, `com.sun.max.config.acme`. However, it is possible to redirect to packages outside the `com.sun.max` namespace. An extension package to be included in Maxine should contain a `Package` class that extends `BootImagePackage`. `BootImagePackage` provides four constructors:

1. `BootImagePackage()`: the default constructor is used to include just those classes in the package containing the `Package` class.
2. `BootImagePackage(boolean)`: this constructor, with a value of true is used to include the classes in the package containing the `Package` class and all sub-packages, recursively. In particular, there is no need for `Package` classes in the sub-packages.
3. `BootImagePackage(String, boolean)`: this redirection constructor includes the classes in the package passed as argument. Sub-packages are only included if the boolean argument is true.
4. `BootImagePackage(String...)`: this constructor serves two purposes. First to include (redirected) packages outside the com.sun.max namespace and, second, to restrict the set of classes that are loaded from a given package. The interpretation of the argument is based on the following patterns:
    - `a.b.c.*`: include all classes from the package `a.b.c`.
    - `a.b.c.**`: include all class from the package `a.b.c` and its sub-packages, recursively.
    - `a.b.c.D`: include the class `a.b.c.D`.

Note that, by design, there is no way to exclude a class from a package, other than by explicitly enumerating all the classes except the one to exclude.

It is legal for a package to be referenced multiple times by different Package constructors. For example, two separate constructors might specify different classes to be included. The final specification is the result of merging the specifications.

Note that the packages generated using `BootImagePackage` are only candidates for inclusion in the image. A package is only actually included if the method `BootImagePackage.isPartOfMaxineVM()` for the `Package` instance returns true. This allows additional controls, such as system properties, to fine tune the inclusion. For recursive and redirected packages, `Package` instances are created automatically by cloning the `Package` that initiated the inclusion, then modifying the package name. Therefore any overridden methods, such as `isPartOfMaxineVM()`, in the including `Package` will be in effect in for all the included packages. However, any actual `Package` classes that do exists in included sub-packages are instantiated and replace the clone.

Configuring the extensions classes

As noted above, adding an extension class does not necessarily cause its methods to be compiled in the boot image. Static fields may also need to be reset if they contain host-specific values and it may also be necessary to re-run the static initializer to create the illusion that class was actually loaded at run-time by the VM. The class `com.sun.max.vm.hosted.Extensions` provided several methods to achieve these goals:

- `resetField(String className, String fieldName)`: reset a static field back to its default value. Note that presently there is no check on the value of either of these arguments, and bad values will be silently ignored.
- `registerClassForReInit(String className)`: run the static initializer on VM startup.
- `registerVMEntryPoint(String className, String methodName)`: register a method as a root for inclusion in the boot image.

7.11.6 Dynamic Extension

A dynamic (VM) extension is packaged in a jar file and loaded by the VM class loader at runtime, just before the main class is loaded, in a very similar way to that for Java agents. The essential difference is that Java agents are considered application extensions and loaded by the system class loader, whereas a VM extension is loaded by the VM classloader. This allows the VM extension classes to reference the VM classes in the boot image and use the extended features of Maxine, e.g., Maxine annotations and systems programming support.

A VM extension jar file must contain a manifest that defines a VMExtension-Class attribute that specifies the class that is the entry point to the extension, e.g.:

```
Manifest-Version: 1.0
VMExtension-Class: com.oracle.max.vm.ext.acme.AcmeExtension
```

This class must define a method with the following signature:

```
public static void onLoad(String args);
```

The VM is fully functional at the time that this method is called but, as for Java agents, the method must return for the startup sequence to proceed. If the extension cannot be resolved (for example, because the extension class cannot be loaded, or because the extension class does not have a conformant onLoad method), the VM will abort. If an `onLoad` method throws an uncaught exception, the VM will abort.

To load the extension at runtime use the `--vmextension:jarpath[=args]` option, `vis`:

```
mx vm --vmextension:acme_vmextension.jar=args ...
```

Multiple extensions may specified and they are processed in the order they appear on the command line. The jar file is appended to the class path used by the VM classloader. If reference needs to be made to additional classes outside the jar file, these may be specified using the optional manifest attribute VM-Class-Path which has a similar specification to Boot-Class-Path for Java agents.

7.12 Memory Management

Automatic memory management has become an essential feature of modern programming languages as it frees programmers from explicit memory management, a time-consuming and error-prone activity.

7.12.1 Meta-circularity and memory management

As a *meta-circular virtual machine*, Maxine can benefit from automatic memory management as well, something that VMs written in lower-level languages such as C or C++ cannot.

An initial design decision in Maxine was to manage (almost) all memory used internally by the VM in the same way as memory used by applications. All internal data structures required for class loading, compilation, verification, etc., are represented as normal heap-allocated Java objects. Although this decision has indeed simplified VM development, an unfortunate consequence is that (internal) VM operations pollute the application heap, with consequences for application performance.

A meta-circular VM can avoid perturbing the application heap and optimize VM performance by exploiting knowledge of the allocation and object lifetime profiles of its internal subsystems. For example, intermediate objects allocated during compilation have a limited lifetime. Once a compilation has finished, only the objects representing the final product remain alive. It would be advantageous to segregate these objects from application objects and to reclaim them with specialized mechanisms that are faster than for general application objects. Similar reasoning can be applied to objects allocated by other sub-systems.

7.12.2 Maxine's current Generational GC

Maxine has recently adopted a simple generational collector implemented by the `GenSSHeapScheme` heap scheme. Details on this new heap scheme and its performance with respect to the original semi-space GC are presented here.

7.12.3 Maxine's semi-space GC

Maxine still includes its original simple semi-space copying collector implemented by `SemiSpaceHeapScheme`. This allows to fall back on a very simple and robust implementation both for experimentation purposes and to diagnose problems.

7.12.4 Next generation GC in Maxine

To address the issues sketched above, we are engaged in the design and implementation of a novel region-based garbage collection sub-system for Maxine. Our intentions are to make Maxine competitive with state of the art GC work and to better address issues specific to meta-circular VMs. In this design

a heap may be composed of fixed-size, possibly non-contiguous regions, in order to favor incremental collections and to support multiple, independently collectible heaps. The GC itself will follow an incremental hybrid mark-sweep approach with policy-driven evacuation.

The long term goals of this effort are to:

- support generational and incremental collection;
- support multiple, independently collectible heaps, with dedicated heaps for VM activities such as compiling, verifying, class loading, etc.;
- foster research on the use and implementation of region-based multiple-heap such as:
  - user-level use of multiple isolated heaps to address pause time issues with large monolithic heaps;
  - investigate GC-heaps that do not requires a contiguous virtual address space; and
  - dynamically attachable object heaps, for example to enable constructs such as shared object memories and persistent pre-populated heaps.

The building blocks for this new GC framework are in place and have been tested with the addition of a pure mark-sweep heap scheme. The mark-sweep heap scheme allows testing of some of the base components of the future region-based garbage collector (namely a tricolor tracing algorithm).

7.13 Management of Code Dependencies

The Management of dependencies from compiled methods to classes, methods and other entities where such dependencies may change and result in some action (e.g. deoptimization) being applied to a compiled method.

7.13.1 Overall Architecture

A dependency is a relationship between a `` `TargetMethod <./Glossary#target-method>`__, that is, the result of a compilation, and an assumption that was made by the compiler during the compilation. The assumption may be any invariant that can be checked for validity at a future time. Assumptions are specified by subclasses of `CiAssumptions.Assumption`. Instances of such classes typically contain references to VM objects that, for example, represent methods, i.e., `RiResolvedMethod`. Note that assumptions at this level are generally specified using compiler and VM independent types, and are defined in a compiler and VM independent project (package). However, there is nothing that prevents a VM specific assumption being defined using VM specific types.

Since an assumption has to be validated any time the global state of the VM changes, for example, a new class is loaded, it must persist as long as the associated `TargetMethod`. To minimize the amount of storage space occupied by assumptions, and to simplify analysis in a concrete VM, validated assumptions are converted to dependencies, which use a densely encoded form of the concrete VM types using small integers, such as `ClassID`.

All assumptions have an associated context class which identifies the class that the assumption affects. For example, the ConcreteSubtype assumption specifies that a class `T` has a single unique subtype `U`. In this case, `T` is defined to be the context class.

The possible set of assumptions and associated dependencies is open-ended. In order to provide for easy extensibility while keeping the core of the system independent, the concept of a

DependencyProcessor is introduced. A DependencyProcessor is responsible for the following:

- the validation of the associated assumption.
- the encoding of the assumption into an efficient packed form
- the processing of the packed form, converting back to an object form for ease of analysis
- supporting the application of a dependency visitor for analysis
- providing a string based representation of the dependency for tracing

7.13.2 Analysing Dependencies

A visitor pattern is used to support the analysis of a Dependencies instance. Recall that each such instance relates to a single TargetMethod, may contain dependencies related to several context classes and each of these may contain dependencies corresponding to several dependency processors.

Since the set of DependencyProcessors is open ended, and a visitor may want to visit the data corresponding to several dependency processors in one visit, implementation class inheritance cannot be used to create a specific visitor. Instead, a two-level type structure is used, with interfaces defined in the specific DependencyProcessor class that declare the statically typed methods that result from decoding the packed form of the dependency. Note that these typically correspond closely to the original CiAssumptions.Assumption but with compiler/VM independent types replaced with Maxine specific types. E.g., RiResolvedType replaced with ClassActor.

7.13.3 Dependencies Visitor

Dependencies.DependencyVisitor handles the aspects of the iteration that are independent of the dependency processors. See Dependencies.DependencyVisitor for more details.

The data for each dependency processor is visited by invoking Dependencies.DependencyVisitor.visit for each individual dependency. This method is generic since it cannot know anything about the types of the data associated with the dependency. The default implementation handles this by calling DependencyProcessor.match which returns dependencyVisitor if the visitor implements the DependencyProcessorVisitor interface defined by the processor that specifies the types of the data in the dependency, or null if not. It then invokes DependencyProcessor.visit with this value, which invokes the typed method in the interface if the value is non-null, and steps the index to the next dependency. Defining DependencyProcessor.visit this way allows a different DependencyProcessorVisitor to be called by an overriding implementation of Dependencies.DependencyVisitor.visit. For example, a visitor that cannot know all the dependency processors in the system, yet wants to invoke the DependencyProcessor.ToStringDependencyProcessorVisitor.

7.13.4 Defining a new Dependency Processor

The first step is to define a new subclass of CiAssumptions.Assumption. If, as is typical, the dependency is used within the optimizing compiler, then this subclass should be defined by adding it to CiAssumptions.

Next define a subclass of DependencyProcessor that will handle this assumption in Maxine, and place it in the com.sun.max.vm.compiler.deps package. Define a nested in-

terface that extends of `DependencyProcessorVisitor` and defines a method with the same arguments as the method in the `CiAssumptions.Assumption` subclass. To support generic tracing of dependencies you should also define a subclass of `DependencyProcessor.ToStringDependencyProcessorVisitor` that implements your interface method(s) and appends appropriate tracing data to the `StringBuilder` variable in `DependencyProcessor.ToStringDependencyProcessorVisitor`.

Define a static final instance of the `DependencyProcessor` subclass, which will cause it to be registered with `DependenciesManager` during boot image generation.

Finally, implement the remaining abstract methods:

- `DependencyProcessor.match`
- `DependencyProcessor.getToStringDependencyProcessorVisitor`
- `DependencyProcessor.visit`

The first two have trivial implementations. The visit method must step over the specific dependency data and, if the `dependencyProcessorVisitor` is not null, invoke the associated method, with the encoded data transformed into the appropriate argument types. Evidently, if the visitor is null, processing related to transforming the encoded data should be avoided.

Automatically generated from `com.sun.max.vm.compiler.deps.package-info`

7.14 Code Eviction in the Maxine VM

Maxine features no interpreter but instead employs only just-in-time compilation. This implies considerable amounts of machine code are created in the course of executing an application. Machine code can become outdated: methods may, after their first execution, later be recompiled by the optimizing compiler, and static class initializers are even executed only once. To address the memory requirement issue, code eviction was introduced to Maxine in Fall 2011 for baseline code, that is, machine code generated by the T1X compiler.

7.14.1 Baseline Code Cache Management

Prior to the introduction of code eviction, Maxine featured two unmanaged code caches, that is, memory areas where machine code is placed. The boot code region contains all machine code belonging to the VM, it is filled during boot image building. The run-time code region was the one where all code generated by either of the JIT compilers went. After introducing code eviction, the boot code and run-time code regions still exist and are unmanaged, but the latter now only contains code generated by the optimizing compiler. One managed code region for baseline code was added. It adopts a semi-space scheme as known from garbage collection. The most important benefit of this scheme is that it implicitly compacts memory upon collection, so that bump-pointer allocation can be applied. The semi-space code region to-space is where newly allocated code is placed (and where code surviving an eviction cycle is moved); its from-space is where code subject to eviction is found.

7.14.2 Code Eviction Workflow

An eviction cycle is triggered when the VM's attempt to allocate space in the baseline code region fails. If that happens, all threads are suspended (eviction is a stop-the-world VM Operation). The workflow is controlled from the `CodeEviction.doIt()` method, and proceeds in the following steps.

Identify Survivors

The eviction logic needs to know which machine code survives the eviction cycle. This takes place in two steps:

1. walk all suspended threads' call stacks and mark the baseline methods currently executing as live. Likewise, their direct callees, if they are baseline methods as well, are marked as live. This ensures that code of methods currently being run in any of the threads will not be evicted, and also that code of methods that are very likely to be invoked again survives.
2. iterate over the current to-space and mark methods as live that need to be protected from eviction for other reasons than being executed or likely to be invoked; namely, methods that have just been compiled but were not yet placed in the baseline code cache (such methods are typically the reason for an eviction cycle to be triggered in the first place) have an invocation counter within optimization threshold and/or a type profile (such methods might soon be recompiled by the optimizing compiler, and the profile information gathered for them should not be lost)

Invalidate Non-live Methods

Invalidation takes place in three steps:

1. the machine code of non-live methods in the baseline code region is overwritten with trap instructions. This is not strictly necessary but greatly helps in debugging
2. all entries in vtables and itables pointing to these methods are invalidated by letting them point to the respective trampolines again. This is facilitated by iterating over the hubs of all loaded classes
3. all direct calls to non-live methods are invalidated by letting them reference trampolines again as well. This implies iterating over all machine code in the three different code regions and checking direct call sites. Direct calls from the boot code region to baseline code are rare, so there is an optimization in place that collects all such calls (they are established at run-time) and thereby avoids iterating over the entire boot code region

Move Live Methods

This is the step where semi-space functionality is actually applied. This affects methods that have not been wiped in the previous step. In particular, this involves the following steps for each live method:

1. Invalidate vtable and itable entries.
2. Copy the method's entire bytes (code and literals arrays) over to to-space.
3. Wipe the machine code and literal arrays as described above.
4. Memoise the old start of the method in from-space, and set new values for its start and end in to-space.
5. Compute and set new values for the code and literals arrays and for the codeStart pointer.
6. Advance the to-space allocation mark by the method's size.
7. Fix direct calls in and to moved code. Direct call sites are relative calls. Hence, all direct calls in moved code have to be adjusted. This is achieved by iterating over all baseline methods (at this point, only methods surviving eviction are affected) and fixing all direct call sites contained therein. Also, direct calls to moved code have to be adjusted. This is achieved by iterating over the optimized and boot code regions and fixing all direct calls to moved code.

8. Compact the baseline code region's target methods array by removing entries for wiped (stale) methods.
9. Fix return addresses on call stacks, and code pointers in local variables. Walk all threads' call stacks once more and fix return addresses that point to moved code. Likewise, fix pointers to machine code held in `CodePointers` in the frames of the methods. This logic makes use of the saved old code start of moved methods.

7.14.3 Tracing and Logging

The eviction algorithm contains copious logging capability using the `VMLogger` mechanism. There are two distinct capabilities; logging the flow of the algorithm and dumping pertinent state of the VM before and after the algorithm executes. Dumping is very verbose and does not place data in the VM log buffer. However, it is defined as an logging operation so that it can be enabled using a consistent mechanism.

The loggable operations are separated into four areas, statistics, algorithmic details, code moving and dumping, with the operations names prefixed by `Stats_`, `Details_`, `Move_` and `Dump`, respectively. The VM option `-XX:+LogCodeEviction` is used to enable logging, with tracing to the log file enabled with `-XX:+TraceCodeEviction`. The options `-XX:+LogCodeEvictionInclude` and `-XX:+LogCodeEvictionExclude` can be used to fine control which options are logged/traced. The operation prefixes can be used in regular expressions to enable all operations in a particular area, for example: `-XX:+LogCodeEvictionExclude=Stats_.*`. Note that dumping must be explicitly enabled with `-XX:+LogCodeEvictionInclude=Dump`.

Automatically generated from com.sun.max.vm.code.package-info

7.15 Object representation in the Maxine VM

This page describes how objects are represented in the Maxine VM runtime. This aspect of the VM is especially interesting for a variety of reasons.

As a *metacircular VM*, most of Maxine's internal data structures, including those used to implement the representation of objects (e.g. *Actors*, *layouts*), are themselves represented uniformly as objects. Some of those objects, however, (see *hubs*) are of an extended type that is not expressible in the Java language, and one kind of object (*static tuples*) cannot be expressed by any type.

As a compiled-only VM, Maxine can only start (*"bootstrap"*) from a synthesized binary boot image (see Garden of Eden configurations). The *boot heap* in particular is a translation of an object graph created during *boot image generation*, a process complicated by the fact that generation is hosted on a standard VM but must deal with objects that cannot be represented in the standard language.

7.15.1 Heap

The general behavior of the heap (memory management, object allocation, garbage collection, etc.) is configurable, defined by a binding to the VM's heap scheme.

The runtime heap contains multiple heap segments. The first heap segment is the boot heap, which is part of the VM's *boot image* and which is pre-populated with the objects that the VM needs in order to begin operation. The boot heap is represented in the standard heap format and is exceptional only in that its objects never move, although they may become permanent garbage.

7.15.2 Object layout

The lowest level details of memory layout of objects (in particular with respect to headers, contents, and pointers) is configured by a binding to the VM's *layout scheme*. The scheme describes layout information for the three kinds of *object representation* described below: tuples, arrays, and hybrids. All memory access to the various parts of an object takes place via a layout scheme.

Of the two runtime layout schemes currently implemented, described below, *OHM layout* is the default binding for CISC architectures (like x86) and *HOM layout* is the default for RISC architectures such as SPARC. A third layout scheme (*Hosted layout*) is specialized for representing objects being prototyped during *boot image generation*, especially important for the kinds of VM objects that cannot be directly represented as standard Java objects.

OHM layout

The OHM object layout scheme, described as Origin-Header-Mixed, is implemented by class `com.sun.max.vm.layout.ohm.OhmLayoutScheme`.

The OHM layout packs tuple objects for minimal space consumption, observing alignment restrictions. Tuple objects have a 2 word header and are laid out as shown below:

```
cell/origin --> +-------------+
                |    class    |  // reference to dynamic hub of class
                +-------------+
                |    misc     |  // monitor and hashcode
       data --> +=============+
                |             |
                :   fields    :  // mixed reference and scalar data
                |             |
                +-------------+
```

OHM array objects have a 3 word header and are laid out as shown below (OHM hybrid objects have a similar header):

```
cell/origin --> +-------------+
                |    class    |  // reference to dynamic hub of class
                +-------------+
                |    misc     |  // monitor and hashcode
                +-------------+
                |   length    |
       data --> +=============+
                |             |
                :  elements   :
                |             |
                +-------------+
```

HOM layout

The HOM object layout scheme, described as Header-Origin-Mixed, is implemented by class `com.sun.max.vm.layout.hom.HomLayoutScheme`.

This layout enables more optimized code on SPARC for accessing array elements smaller than a word. The origin points at array element 0, so there is no need to perform address arithmetic to skip over the

header. On the other hand, this layout requires reading memory for converting between cell and origin addresses, since they are not the same (as they are for *OHM layout*).

The HOM layout packs tuple objects for minimal space consumption, observing alignment restrictions. Tuple objects have a 2 word header and are laid out as shown below.

```
       cell --> +-------------+
                |    misc     |  // monitor and hashcode
                +-------------+
                |   class     |  // reference to dynamic hub of class
origin/data --> +=============+
                |             |
                :   fields    :  // mixed reference and scalar data
                |             |
                +-------------+
```

Array objects have a 3 word header and are laid out as shown below (HOM hybrid objects have a similar header):

```
       cell --> +-------------+
                |   length    |
                +-------------+
                |    misc     |  // monitor and hashcode
                +-------------+
                |   class     |  // reference to dynamic hub of class
origin/data --> +=============+
                |             |
                :  elements   :
                |             |
                +-------------+
```

Hosted layout

The Hosted object layout scheme is not designed for VM runtime, but rather for the object prototyping phase of boot image generation. The generation machinery runs hosted on a standard Java VM and creates a prototype boot heap that will eventually be translated into the binary format of the target platform, and written into the boot image. This layout scheme is implemented by class `com.sun.max.vm.layout.hosted.HostedLayoutScheme`.

7.15.3 Object representation

There are exactly three low-level memory representations in the Maxine heap: *Tuple representation* (for Java object instances), *Array representation* (for Java array instances), and *Hybrid representation* (for Maxine hubs). Memory access to the parts of these three representations is mediated through a *layout scheme*. Types and other aspects of object contents are defined by the `ClassActor` instance that represents type of the object being represented.

Tuple representation

A Maxine tuple is a memory representation that combines a two-word header plus a collection of named values (fields). The names, types, and locations of the values are defined by an instance of class `TupleClassActor`.

As with all Maxine object representations, the first word of the tuple header points at the *Dynamic hubs* for the class. The second (misc) word is used for a variety of purposes, including hash code and locking information.

The tuple memory representation is used to represent standard Java class instances in the heap. Note that *Static tuples* are also represented this way, even they are not ordinary class instances and have no type.

Array representation

A Maxine array is a memory representation that combines a three-word header plus some fixed number of values of identical type. The type of the array elements is defined by an instance of class `ArrayClassActor`.

As with all Maxine object representations, the first word of the array header points at the *Dynamic hubs* for the class. The second (misc) word is used for a variety of purposes, including hash code and locking information. The third word holds the number of elements contained in the array.

The array memory representation is used to represent standard Java arrays in the heap.

Hybrid representation

A Maxine hybrid is a memory representation that combines a three-word header, a collection of named values (fields), and an array of words. The names, types, and location of the field values, together with information about the arrays, are defined by an instance of class `HybridClassActor`. Although hybrids are represented uniformly as instances of a class, they are classes that cannot be expressed in standard Java.

As with all Maxine object representations, the first word of the hybrid header points at the *Dynamic hubs* for the class. The second (misc) word is used for a variety of purposes, including hash code and locking information. The third word holds the number of words contained in the array.

The hybrid memory representation is used to represent Maxine *Hubs* in the heap, even though hubs are not standard Java class instances and cannot be described with standard Java types.

7.15.4 Actors

Specific information about the contents of heap instances (tuples, arrays, and hybrids) is represented uniformly using Java type information, represented in the form of *class actors*. Class actors are themselves instances (represented as tuples) in the heap of the three types `TupleClassActor`, `ArrayClassActor`, and `HybridClassActor` respectively.

7.15.5 Hubs

A hub is a *Hybrid representation* instance holding information, derived from a *class actor*, that must be immediately accessible (one memory hop) from each class instance. That is, a hub is what is pointed to from the (logical) class word of an object's header. A hub corresponds to a TIB in the Jikes RVM.

Hubs hold the vtables and itables used for efficient method dispatch. They also hold all the information needed when a garbage collector visits each instance, for example the size and reference map for the instance, avoiding the need to reference any further objects, which could themselves be subject to collection.

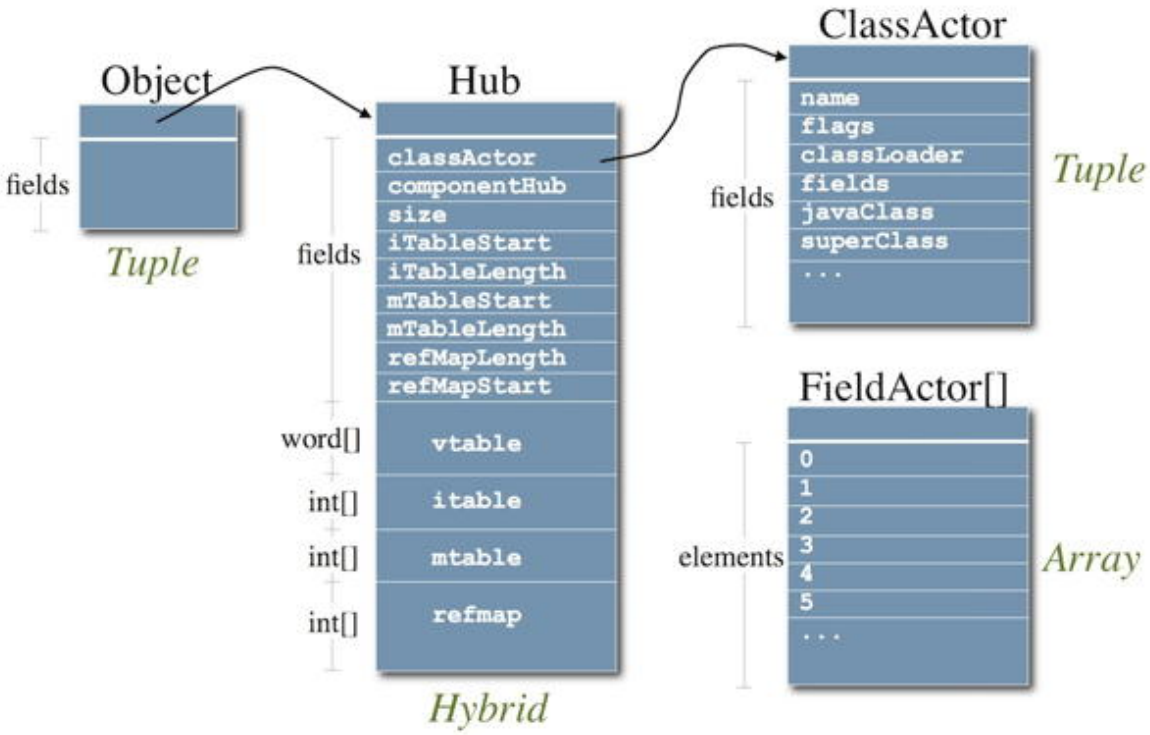
Hubs contain both named fields and embedded arrays and thus cannot be represented as a ordinary Java objects. They are instead represented as *Hybrid representation*, and their contents are described uniformly with a class actor (describing a class not expressible in standard Java) of type `HybridClassActor`.

There are two kinds of hubs, distinguished by the context of their use: *Dynamic hubs* are pointed to by class instances, and *Static hubs* are pointed to by *Static tuples*.

Dynamic hubs

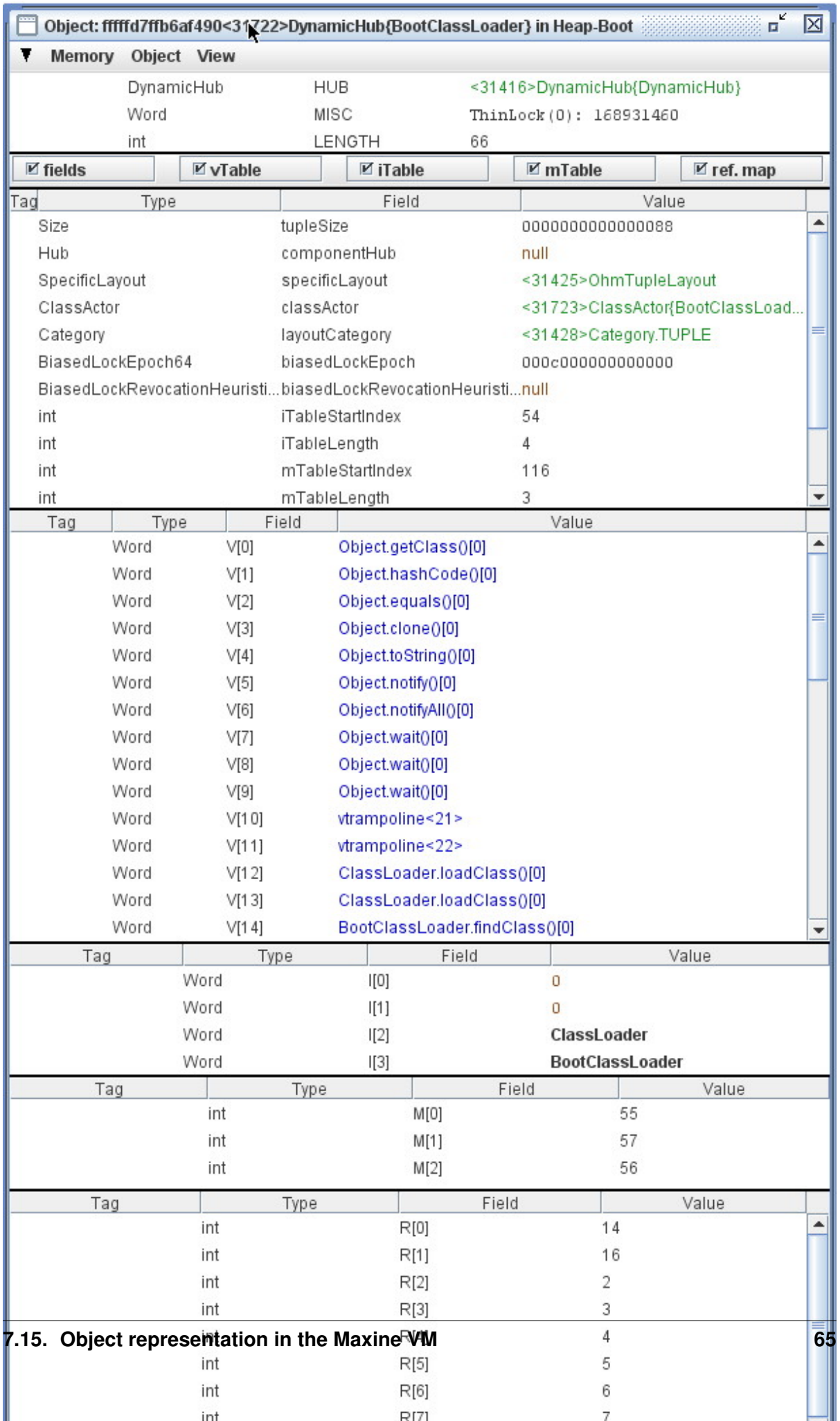
Every `ClassActor` in the VM holds a reference to the dynamic hub (an instance of class `com.sun.max.vm.actor.holder.DynamicHub`) for the class it represents. Every instance of that class in the heap contains (in its header) a reference to that dynamic hub.

The following figure depicts the relationships among a class instance, the dynamic hub for the class, and the `ClassActor` for the type. The figure also demonstrates the three kinds of representation in the heap: tuples, arrays, and hybrids.



The following screen snapshot shows how the dynamic hub for class `com.sun.max.vm.type.BootClassLoader` appears in an *Object Inspector View* in the *Maxine Inspector*. It is displayed as a hybrid object, with special display machinery for viewing the embedded arrays. The inspector's frame header identifies it as `DynamicHubBootClassLoader`, meaning *the ''DynamicHub'' associated with class ''BootClassLoader''*.

Note also that the hub pointer for this instance of `DynamicHub` points to another `DynamicHub` which is described as *the ''DynamicHub'' associated with class ''DynamicHub''*. In other words, the hub pointer of that hub points at itself: it participates in its own implementation and closes the hub recursion loop.

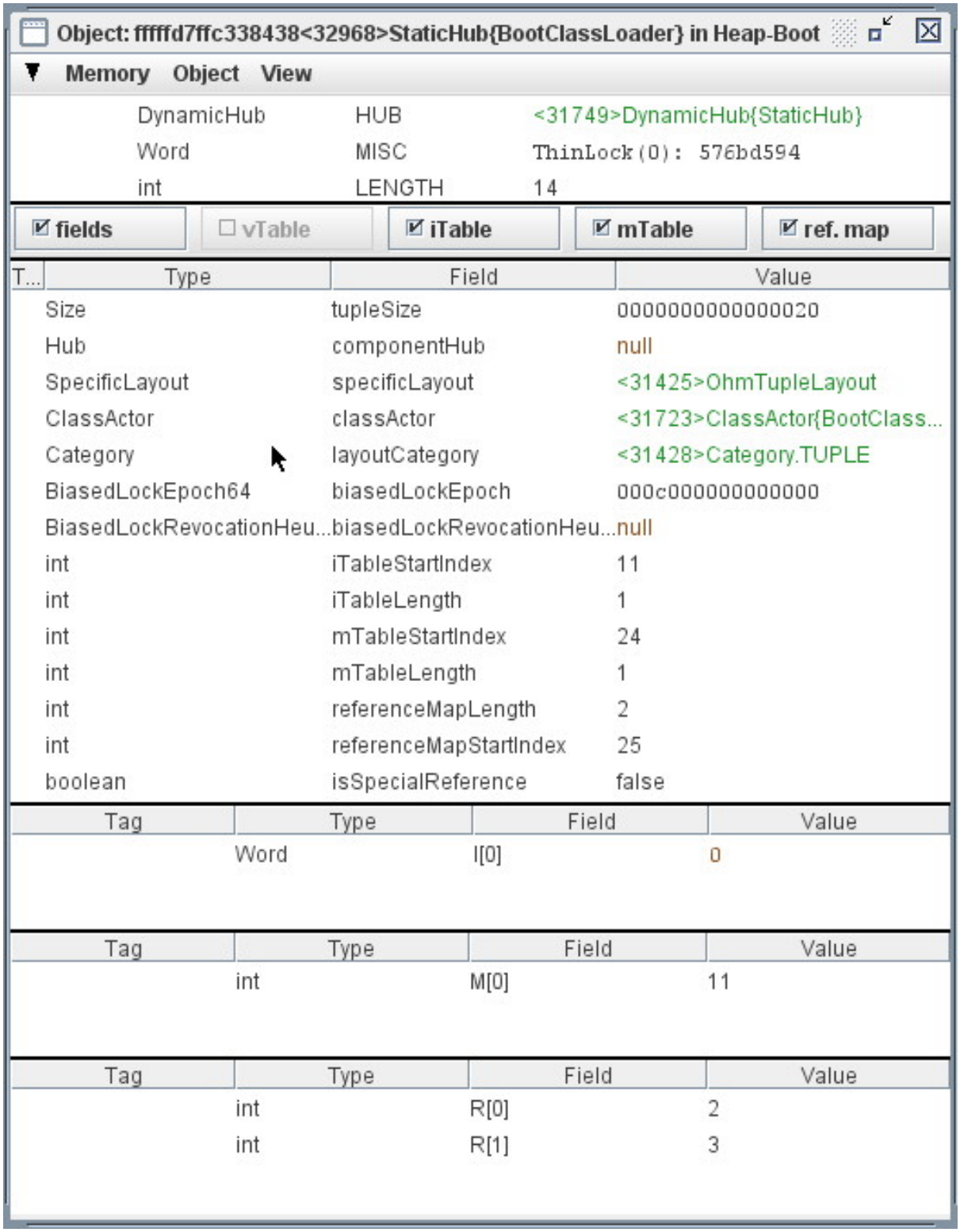

Object: fffffd7ffb6af490<31722>DynamicHub{BootClassLoader} in Heap-Boot
Memory Object View
DynamicHub HUB <31416>DynamicHub{DynamicHub}
Word MISC ThinLock(0): 168931460
int LENGTH 66
fields vTable iTable mTable ref. map
Tag Type Field Value
Size tupleSize 0000000000000088
Hub componentHub null
SpecificLayout specificLayout <31425>OhmTupleLayout
ClassActor classActor <31723>ClassActor{BootClassLoad...
Category layoutCategory <31428>Category.TUPLE
BiasedLockEpoch64 biasedLockEpoch 000c000000000000
BiasedLockRevocationHeuristi... biasedLockRevocationHeuristi... null
int iTableStartIndex 54
int iTableLength 4
int mTableStartIndex 116
int mTableLength 3
Tag Type Field Value
Word V[0] Object.getClass()[0]
Word V[1] Object.hashCode()[0]
Word V[2] Object.equals()[0]
Word V[3] Object.clone()[0]
Word V[4] Object.toString()[0]
Word V[5] Object.notify()[0]
Word V[6] Object.notifyAll()[0]
Word V[7] Object.wait()[0]
Word V[8] Object.wait()[0]
Word V[9] Object.wait()[0]
Word V[10] vtrampoline<21>
Word V[11] vtrampoline<22>
Word V[12] ClassLoader.loadClass()[0]
Word V[13] ClassLoader.loadClass()[0]
Word V[14] BootClassLoader.findClass()[0]
Tag Type Field Value
Word I[0] 0
Word I[1] 0
Word I[2] ClassLoader
Word I[3] BootClassLoader
Tag Type Field Value
int M[0] 55
int M[1] 57
int M[2] 56
Tag Type Field Value
int R[0] 14
int R[1] 16
int R[2] 2
int R[3] 3
int R[4] 4
int R[5] 5
int R[6] 6
int R[7] 7

Static hubs

There is exactly one kind of instance, represented as a *Tuple representation* in the heap, that cannot be treated uniformly by the VM's type information: a static tuple. A static tuple is unique in that cannot be described by a type, so it has no ClassActor that describes its type and must be treated exceptionally wherever types matter.

Every `ClassActor` in the VM holds a reference to a static tuple, which holds values of the class (static) variables for the class. Each `ClassActor` also holds a reference to the static hub, an instance of class `com.sun.max.vm.actor.holder.StaticHub`, to which the header of the static tuple points. This specialized hub, to which only the static tuple points, allows uniform treatment by GC.

The following screen snapshot shows how the static hub for class `com.sun.max.vm.type.BootClassLoader` appears in an *Object Inspector View* in the *Maxine Inspector*. It is displayed as a *Hybrid representation*, with special display machinery for viewing the embedded arrays. The inspector's frame header identifies it as `StaticHubBootClassLoader`, meaning *the ''StaticHub'' associated with class ''BootClassLoader''.*

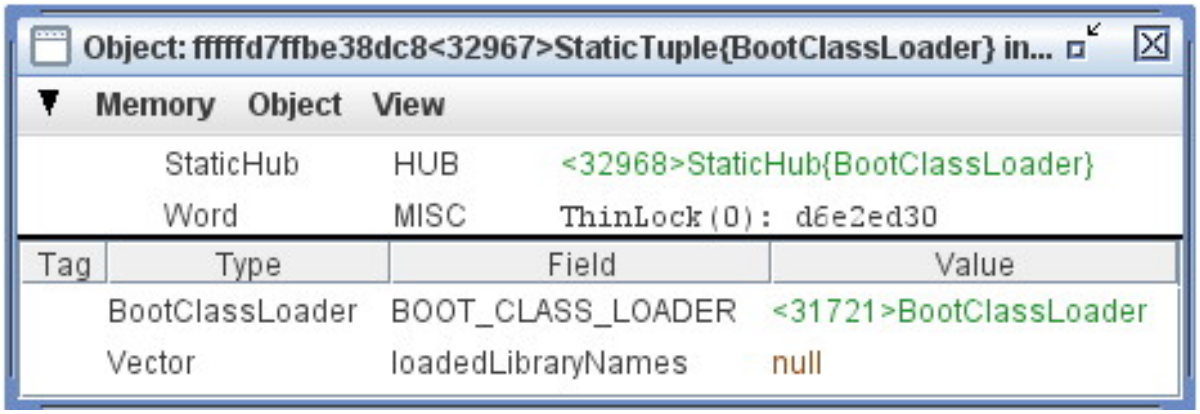


7.15.6 Static tuples

A static tuple is special kind of heap object that holds the class variables (static fields) for a class in the VM. Although it appears superficially as an ordinary *Tuple representation*, with named fields, it is unique within the heap in that it has no type at all: think of it as a singularity in the VM's type system (a byproduct of the VM's *metacircularity*). In practice, this means that there is no `ClassActor` describing any static tuple; they must be treated using implicit knowledge of their structure.

On the other hand, static tuples are represented the same as other tuples in heap memory, and are amenable to ordinary garbage collection without special handling. This is done by having a special kind of *Static hubs* to which they point.

The following screen snapshot shows how the static tuple for class `com.sun.max.vm.type.BootClassLoader` appears in an *Object Inspector View* in the *Maxine Inspector*. It is displayed as an ordinary tuple, but the frame header identifies it as `StaticTupleBootClassLoader` and its hub pointer refers to an instance of `StaticHub`.



7.16 Schemes: Interfaces for Maxine VM Configuration

The Maxine VM is designed to be highly configurable, and it exploits standard Java language features as much as possible to accomplish this. Maxine schemes are Java interfaces that define the interaction between a subsystem and the rest of the VM.

This design encourages the creation of alternate implementations of those schemes, for example to achieve different performance characteristics. It also enables the creation of specialized implementations for development and testing, for example monitors that have no effect, a heap that doesn't collect, or any implementation with extensive internal checking and tracing.

The current design of schemes where by all logic is expressed purely in Java code is heavily tied to the use of Java snippets which, in turn, are only supported by the C1X compiler. Given that this compiler is scheduled to be deprecated and replaced by the Graal compiler, the design of schemes will be re-visited to work with Graal.

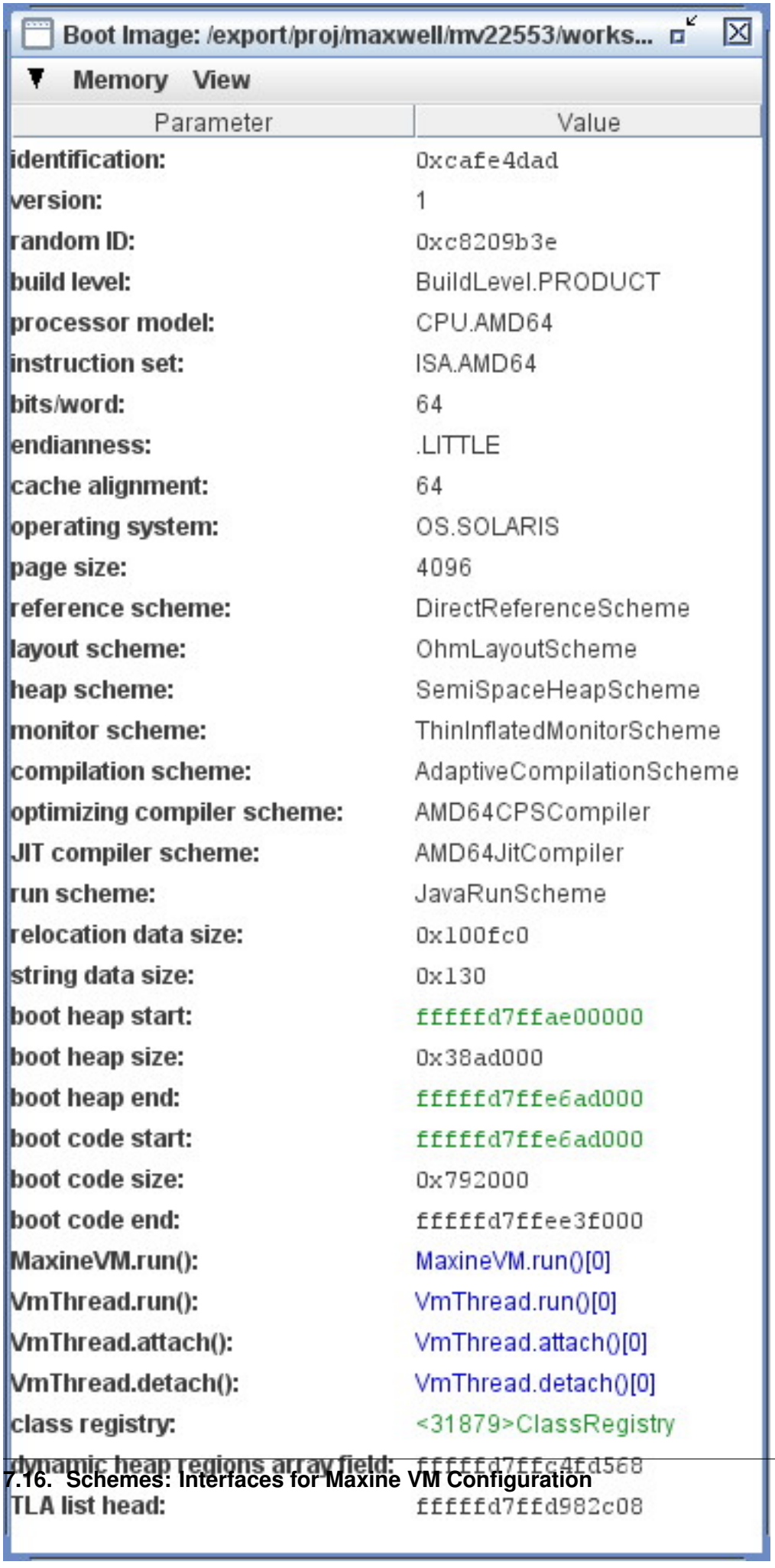
7.16.1 VM configuration

All Maxine schemes extend the interface `com.sun.max.vm.VMScheme`.

A specific implementation of a scheme is by convention located in its own package and constitutes a Maxine Package. It is documented, following Javadoc convention, by a file named `package-info.java` in the package directory. It may also contain a class named `Package`, which may contain additional configuration information.

A complete VM configuration includes bindings to a specific implementation of each scheme, specified as command line options, processed by the class `VMConfigurator`, and represented during boot image generation and VM runtime by an instance of class `VMConfiguration`.

The Boot Image Inspector view in the *Maxine Inspector* displays, among the other information about the specific boot image being viewed, the specific bindings for each of Maxine's schemes.


Boot Image: /export/proj/maxwell/mv22553/works...
Memory View
Parameter
Value
identification: 0xcafe4dad
version: 1
random ID: 0xc8209b3e
build level: BuildLevel.PRODUCT
processor model: CPU.AMD64
instruction set: ISA.AMD64
bits/word: 64
endianness: .LITTLE
cache alignment: 64
operating system: OS.SOLARIS
page size: 4096
reference scheme: DirectReferenceScheme
layout scheme: OhmLayoutScheme
heap scheme: SemiSpaceHeapScheme
monitor scheme: ThinInflatedMonitorScheme
compilation scheme: AdaptiveCompilationScheme
optimizing compiler scheme: AMD64CPSCompiler
JIT compiler scheme: AMD64JitCompiler
run scheme: JavaRunScheme
relocation data size: 0x100fc0
string data size: 0x130
boot heap start: ffffffd7ffae00000
boot heap size: 0x38ad000
boot heap end: ffffffd7ffe6ad000
boot code start: ffffffd7ffe6ad000
boot code size: 0x792000
boot code end: ffffffd7ffee3f000
MaxineVM.run(): MaxineVM.run()[0]
VmThread.run(): VmThread.run()[0]
VmThread.attach(): VmThread.attach()[0]
VmThread.detach(): VmThread.detach()[0]
class registry: <31879>ClassRegistry
dynamic heap regions array field: ffffffd7ffc4fd568
TLA list head: ffffffd7ffd982c08

7.16.2 Scheme Initialization

Each scheme implementation is notified by a call to the method `initialize(MaxineVM.Phase phase)` when the VM enters different phases of its lifecycle. These phases are defined in the enum Phase in class `com.sun.max.vm.MaxineVM`.

Phase `BOOTSTRAPPING` is assigned during *boot image generation* when the scheme implementation is loaded, something of a misnomer since the term *bootstrapping* is generally used to describe the startup sequence of the VM.

The following list describes the initialization calls that each scheme will receive, in order, during startup, along with the presumed state of the VM in each instance:

- `PRIMORDIAL` VM code has started executing, but many features do not work yet.
- `PRISTINE` Java thread synchronization has become operational, but may not do anything yet.
- `STARTING` the VM is functional (i.e. threads and heap), but the JDK is not yet operational.
- `RUNNING` any necessary re-initializion of JDK classes is complete, system properties have been processed, all pure Java language features are operational, and the VM is about to start executing application code.
- `TERMINATING` the VM is about to terminate, many VM features have shut down, and this the last chance to interpose (but with very limited VM functionality).

7.16.3 Object Layout (LayoutScheme)

An implementation of the scheme defined by interface `com.sun.max.vm.layout.LayoutScheme` configures how objects are represented in memory, including header and fields. See *Object layout* for more details.

7.16.4 Object References (ReferenceScheme)

An implementation of the scheme defined by interface `com.sun.max.vm.reference.ReferenceScheme` configures how objects are accessed for mutator use, for example direct pointers or handles. The default binding is `DirectReferenceScheme`.

Note also that the *Inspector*, which runs in a separate process than the VM, is able to reuse a considerable amount of VM code in a uniform way by creating a pseudo-configuration and binding a custom implementation of `ReferenceScheme` that encapsulates a boxed address in the address space of the VM process.

7.16.5 Heap allocation and garbage collection (HeapScheme)

An implementation of the scheme defined by interface `com.sun.max.vm.heap.HeapScheme` provides basic heap memory management, object allocation, and garbage collection.

Different implementations may include very different kinds of collectors. For example, the default binding is `GenSSHeapScheme`, which implements a simple generational collector. Another binding is `SemiSpaceHeapScheme`, which implements a straightforward semi-space collector. A third heap scheme is the `MSHeapScheme`, which implements a pure mark-sweep algorithm.

For certain kinds of testing it can be useful to bind implementations with limited or specialized functionality, for example with an implementation that allocates but never collects or one with extensive heap checking or tracing.

7.16.6 Thread Synchronization (MonitorScheme)

An implementation of the scheme defined by interface `com.sun.max.vm.monitor.MonitorScheme` supports synchronization in the VM.

The interface represents an abstraction of monitors. It specifically includes translation of the `monitorenter` and `monitorexit` bytecodes, as well as the implementation of `wait` and `notify` methods. Implementations might include thin locks, biased locking, hybrids, etc.

Some experimentation has already been done in this area, and the implementation currently in use is part of a framework for "n-modal" locking schemes, a generalization of bimodal (as in JikesRVM) and trimodal designs. The framework is implemented by abstract class `ModalMonitorScheme` in package `com.sun.max.vm.monitor.modal.schemes`. The default binding at present (implemented by class `ThinInflatedMonitorScheme`) transitions between thin locks and inflated native monitors.

For certain kinds of testing it can be useful to disable monitor checking completely; this can be done by binding the class `IgnoreMonitorScheme` into the VM configuration.

7.16.7 VM startup sequence (RunScheme)

An implementation of the scheme defined by interface `com.sun.max.vm.run.RunScheme` is invoked by the VM after it has started basic services and is ready to set up and run a language environment such as Java or some other language.

The default binding is the standard Java runtime: `JavaRunScheme` (in package `com.sun.max.vm.run.java`) starts up normal JDK services (a somewhat delicate piece of business), and then loads and runs a user-specified Java class.

7.16.8 Compiler strategy (CompilationBroker)

The class `com.sun.max.vm.compiler.CompilationBroker` implements an adaptive compilation system with multiple compilers with different compilation time / code quality tradeoffs. It encapsulates the necessary infrastructure for recording profiling data, selecting what and when to recompile, etc.

The class `CompilationBroker` can be subclassed by using the `max.CompilationBroker.class` system property with the boot image generator.

This is done as follows:

```
max -J/a-Dmax.CompilationBroker.class=com.acme.MyCompilationBroker image ..
↪.
```

Note: that the `CompilationBroker` is going to be removed as well and be replaced by the JVM Compiler Interface (JVMCI).

7.17 Snippets in the Maxine VM

Snippets are designed to provide a clean separation between the runtime system and compiler in a JVM. A clean separation allows either component to be modified or extended independently in the pursuit of features or optimization. Furthermore, snippets are designed to isolate the implementation of runtime features from each other. For example, the implementation of virtual dispatch need not be aware of the object model or memory management implementation. The rest of this page provides the implementation details of snippets in the Maxine VM and how they measure up against these design goals. This is done by focusing on how the `invokevirtual` bytecode is implemented via snippets. This is a non-trivial bytecode whose implementation depends on numerous runtime features.

When implementing any given JVM bytecode instruction in a compiler, it is useful to think of the implementation in two parts: the JVM specification semantics and the VM runtime details. The compiler has little or no flexibility with respect to JVM specification semantics; developers write code with these semantics in mind and expect the JVM to be compliant. The runtime details however are completely at the discretion of the VM implementer. Should vtables be used? If so, how are they laid out? How can runtime information inform which optimizations (such as inlining) should be performed or even reversed (de-optimized)?

With this context in mind, one description of snippets is that they are simply pieces of Java source code that express the (partial) semantics and implementation of a bytecode instruction. There are typically a number of snippets which, together, comprise the implementation of a single bytecode instruction. Continuing with the `invokevirtual` example, the following summarizes its specification:

1. Resolve the method denoted by a symbolic reference in a constant pool.
2. Select the target method to be invoked based on the resolved method and the type of the receiver (i.e. this at the call site).
3. Call the selected target method.

The first two steps are expressed by the following snippets:

```
@SNIPPET
@INLINE(afterSnippetsAreCompiled = true)
public static VirtualMethodActor resolveVirtualMethod(ResolutionGuard␣
↪guard) {
    if (guard.value == null) {
        resolve(guard);
    }
    return UnsafeCast.asVirtualMethodActor(guard.value);
}
```

```
@INLINE(afterSnippetsAreCompiled = true)
public static Word selectNonPrivateVirtualMethod(Object receiver,␣
↪VirtualMethodActor declaredMethod) {
    final Hub hub = ObjectAccess.readHub(receiver);
    return hub.getWord(declaredMethod.vTableIndex());
}

@SNIPPET
@INLINE(afterSnippetsAreCompiled = true)
public static Word selectVirtualMethod(Object receiver, VirtualMethodActor␣
↪declaredMethod) {
    if (declaredMethod.isPrivate()) {
        // private methods do not have a vtable index, so directly compile␣
↪and link the receiver.
```

(continues on next page)

(continued from previous page)

```
        return CompilationScheme.Static.compile(declaredMethod,␣
↪CallEntryPoint.VTABLE_ENTRY_POINT);
    }
    return selectNonPrivateVirtualMethod(receiver, declaredMethod);
}
```

The second step (i.e. `selectVirtualMethod`) yields the machine code address (of the target method) to be called. The compiler is expected to know how to emit a call (given a target address and signature), and so a snippet is not used for this step.

The last point exemplifies an aspect of a compiler built around snippets. The compiler is expected to be able to translate certain operations intrinsically. This includes all primitive operations such integer, float and long arithmetic, local control flow constructs as well as the aforementioned capability for emitting a call given a target address and signature. In Maxine, these operations are called compiler ***builtins***. Like snippets, they are also expressed as Java methods. For example, here is the builtin for integer addition:

```
public static class IntPlus extends JavaBuiltin {
    @BUILTIN(builtinClass = IntPlus.class)
    public static int intPlus(int a, int b) {
        return a + b;
    }
}
```

The body of the `intPlus` method above exists solely for the purpose of folding (i.e. compile-time evaluation) as well as IR interpretation. The compiler (backend) knows how to emit machine code whenever it comes across a call to a builtin (i.e. any method with the `@BUILTIN` annotation).

7.17.1 VM feature isolation

Before drilling down to the details of how snippets are built and consumed by the compiler, it's worth using the invokevirtual example to demonstrate how snippets isolate the implementation details of VM features from one other. Consider the following line in the `ResolveVirtualMethod` snippet:

```
if (guard.value == null) {
```

Depending on the VM configuration, a number of VM features (detailed below) are exercised by the read-access of the `value` field from the `guard` object. While reading these, keep in mind that not one of them is explicitly present in the snippet source code.

- **Object model:** An object model specifies how fields, array elements and object metadata are layed out in the memory allocated for an object. The object model in the Maxine VM is a configurable component represented by the `LayoutScheme` interface. There are currently two different object model implementations in Maxine. With respect to snippets, the point to note is that when switching between object models, there is no need to modify the code of the `ResolveVirtualMethod` and `SelectVirtualMethod` snippets.
- **Garbage collector barriers:** If the VM is configured with a garbage collector that uses read-barriers, then using a barrier (if necessary) for the read of the value value is solely the responsibility of the snippet implementing reading of reference fields.
- **Garbage collector handles:** The compiler tracks the types of Java variables and generates the appropriate reference maps such that a GC can find all the object references in method activation.

- **Object references:** Maxine includes two (related) abstractions for specifying how object references are implemented. The first, represented by the `ReferenceScheme` interface, encapsulates the operations that can be applied to an object reference (a value of type `java.lang.Object`) such as reading or writing a char from a reference at a given offset. This abstraction has support for read or write barriers and so is used when compiling mutator (i.e. non-GC) code. The second abstraction, represented by the GripScheme interface, has the same operations as the first except that it omits any notion of barriers. A `GripScheme` deals with values of type `Grip` and is used when implementing a garbage collector. Typically, an implementation of a `ReferenceScheme` is bound to an implementation of a `GripScheme`. The default implementation of `GripScheme` is `DirectGripScheme` which treats object references as direct memory pointers. However, alternative `GripScheme` implementations could be used to implement:
  - compressed oops
  - indirect object references via a handle table
  - object references on a system that has hardware support for objects

7.17.2 IR Notation

The following sections include compiler IR examples. To aid comprehension of these examples, the IR notation is informally described here.

The IR is composed of values, operations and procedure/function calls. Calls are composed of a target followed by a set of (comma separated) arguments enclosed by '(' and ')'. A target enclosed by '<' and '>' is a builtin.

Values are named variables (e.g. method), constant objects prefixed with '@' (e.g. @GUARD_FOR_NAME) or primitive constants (e.g. 32).

All values and targets are typed. The type is indicated by a '#' suffix followed by one of the type characters in this table:

Character	Description	Bit width
R	an object reference	width of machine word
W	an unsigned word	width of machine word
I	int	32
J	long	64
F	float	32
D	double	64

The IR also has expressions, assignments, control flow and return constructs that should be self explanatory to anyone familiar with Java.

7.17.3 Using snippets

So how does the compiler actually use snippets when translating bytecode? The basic idea is that the compiler translates each snippet into an IR (intermediate representation) graph which is stored in a compiler-internal data structure. The issue of how the compiler initializes the collection of IR snippets is described in the next section.

Here is an example of the IR that may be produced for the `ResolveVirtualMethod` and `SelectVirtualMethod` snippets:

```
resolveVirtualMethod(guard#R)#R {
    value#R := <readReferenceAtIntOffset>#R(guard#R, 24#I);
    if (value#R == null#R) {
        resolve#V(guard#R);
    }
    result#R := <readReferenceAtIntOffset>#R(guard#R, 24#I);
    return#R result#R;
}

selectVirtualMethod(rcvr#R, method#W)#R {
    flags#I := <readIntAtIntOffset>#R(method#R, 32#I);
    tmp#I := <intAnd>#I(flags#I, 2#I);
    if (tmp#I == 0) {
        result#W := vtableDispatch#W(rcvr#R, method#R);
    } else {
        result#W := compile#W(method#R, @VTABLE_ENTRY_POINT#R);
    }
    return#W result#W;
}
```

When compiling other (non-snippet) methods, the front-end of the compiler responsible for parsing bytecodes produces IR by weaving hand-crafted IR with the relevant snippet IR. For example, consider the following Java source code method:

```
public String toString() {
    return name();
}
```

The bytecode produced by javac for this method is:

```
aload_0
invokevirtual "name()"
areturn
```

When compiling this method, the compiler will weave in the pre-built IR for the `ResolveVirtualMethod` and `SelectVirtualMethod` snippets to produce the following:

```
asString(this#R)#R {
    value#R := <readReferenceAtIntOffset>#R(@GUARD_FOR_NAME#R, 24#I);
    if (value#R == null#R) {
        resolve(guard#R);
    }
    method#R := <readReferenceAtIntOffset>#R(@GUARD_FOR_NAME#R, 24#I);
    flags#I := <readIntAtIntOffset>#I(method#R, 32#I);
    tmp#I := <intAnd>#I(flags#I, 2#I);
    if (tmp#I == 0) {
        address#W := vtableDispatch#W(this#R, method#R);
    } else {
        address#W := compile#W(method#R, VTABLE_ENTRY_POINT#R);
    }
    result#R := <call>#R(address#W, this#R);
    return#R result#R;
}
```

Note that this is the code produced when the compiler has determined that the name method has not yet been resolved. To determine that a method has been resolved, a compiler based on snippets can rely upon folding and inlining during compilation. For this example, the guard object is a compile time constant wrapping a resolved symbolic reference to method. Constant propagation combined with inlining and folding will therefore reduce the above IR to a vtable dispatch:

```
asString(this#R)#R {
    hub#R := <readReferenceAtIntOffset>#R(this#R, 0#I);
    address#W := <builtinGetWord>#W(hub#R, 24#I, 64#I);
    result#R := <call>#R(address#W, this#R);
    return#R result#R;
}
```

Here is the source for `vtableDispatch`:

```
@INLINE
public static Word vtableDispatch(Object receiver, VirtualMethodActor␣
↪declaredMethod) {
    Hub hub = ObjectAccess.readHub(receiver);
    return hub.getWord(declaredMethod.vTableIndex());
}
```

Note that the vtable dispatch logic also benefits from the VM feature isolation offered by snippets. That is, it does not explicitly mention how to read a hub from an object - it just calls a method that does it (which in turn is inlined).

While this compilation strategy produces optimal and correct code, its performance can suffer if the pursuit of non-redundancy is uncompromising. The mechanism by which folding is performed in the CPS compiler is to call Java methods via reflection. The overhead of reflection is significant:

- the IR values must be unboxed from their IR boxing types and then re-boxed to their Java boxing types
- the reverse unboxing and reboxing is required for the return value
- a new array of arguments is constructed by stripping the continuation arguments from the CPS call IR construct
- there is no chance for the compiler to inline the call and elide the boxing as these reflective calls are made from general purpose folding logic
- reflection mandates type checking for all arguments, something that is redundant with the type checking performed by the compiler itself

To address these performance concerns, the compiler can intrinsify some of the logic expressed in the snippets. For example, it can do the resolution check itself. It can also determine if a resolved method is private in which case it would simply prefer to use the `vtableDispatch` method as a snippet. In general, there's a need to revisit how the logic is split between the compiler and snippets so that compiler performance is maximized while benefits of snippets are not lost.

7.17.4 Bootstrapping snippets

As seen in the previous section, the compiler uses pre-built IR snippets when compiling Java bytecode methods. We've also shown how snippets are expressed as Java bytecode methods (derived from Java source code). These two facts combined represent a cyclic dependency between the compiler and the pre-built snippets. Snippets may also have cyclic dependencies among themselves. For example, the

Java source code for the `ResolveVirtualMethod` and `SelectVirtualMethod` snippets use virtual method invocation themselves. In fact, almost all snippets depend on virtual method invocation. These cyclic dependencies pose a bootstrap problem to the compiler implementer.

The general strategy to resolve all of these circular dependencies is to prepare the snippets using two passes over all snippets:

1. The first pass (***snippet creation***) translates each snippet to IR without engaging in any optimizations at all except mandatory inlining as directed by an `@INLINE` annotation. The snippets are carefully crafted in such a way that they can make use of each other on an inlining basis, practically using other snippets as macros.
2. The second pass (***snippet optimization***) optimizes the output of the first pass and stores the optimized IR in a table.

A predicate is maintained by the compiler indicating whether the second phase has completed or not. This information is used by the compiler to interpret the `afterSnippetsAreCompiled` flag of the `@INLINE` annotation. When the annotation is present at a method declaration, then a call to the method is inlined *iff* its compilation occurs after the second pass. This mechanism allows snippets to contain method calls so that bootstrapping the snippets themselves bottoms out. Nevertheless these calls can later be inlined after all snippets are available, while compiling other code. In other words, pre-built snippet IR may not be fully optimized, but once woven into user code, they are subject to full optimization.

7.17.5 Annotations

The Java code for snippets relies on the following annotations, which serve as pragmas for the compiler:

- `@SNIPPET`: Denotes the entry point for a snippet.
- `@FOLD`: The annotated method must must have no arguments (apart from the implicit this if it is a not `static` method). If the method is `static`, it is evaluated unconditionally by the compiler. If the method is not `static`, it will be evaluated by the compile whenever its receiver is a compile time constant.
- `@INLINE`: The annotated method is inlined by the compiler. If the `afterSnippetsAreCompiled` flag has the default value (i.e. `false`), then the inlining is performed unconditionally. Otherwise, inlining is conditional upon the snippet bootstrapping phase as described above.
- `@NEVER_INLINE`: The annotated method is never inlined by the compiler. In the context of snippets, this is useful for denoting a slow path when generating code. That is, code the is rarely expected to be called and so should not be inlined in the method being compiled.

7.17.6 Evaluation

1. **Performance**: To what extent do snippets affect the runtime and/or code quality of a compiler?
    - [STRIKEOUT:Snippets in the Maxine VM are supported and used by the CPS compiler, the only compiler currently capable of bootstrapping Maxine. The CPS compiler was co-designed with the snippet mechanism. Unfortunately, the performance of this compiler is sub-optimal both in terms of compilation speed and quality of compiled code. Given the many factors affecting the quality of a compiler (choice of IR, register allocation algorithm employed, optimizations performed, memory usage during compilation, etc.), it is hard measure the impact of snippets on the compiler's performance. To perform a meaningful assess-

ment of snippets, one ideally needs to start with a compiler that is not based on snippets and then modify it to use snippets. By doing so, once can measure the extent to which snippets improve/degrade the runtime and/or code quality produced by a compiler. In addition, this experiment will reveal the architectural impact of making a compiler snippet aware. That is, to what extent do snippets complicate (or simplify) a compiler's design. Once the C1X compiler is integrated into Maxine, it will form the basis for such an experiment and thus provide an answer to the performance question.] (outdated)

2. **Expressiveness**: How easy is it to express/comprehend the semantics of a bytecode instruction?
    - Being written in Java source code, snippets can mostly be as easily written and comprehended as any other piece of Java code. The qualification is that one needs to be very aware of the potential for causing infinite recursion. For example, when implementing the athrow bytecode, it is important not to include any code that explicitly throws an exception. Fortunately, infinite recursion is usually fail-fast and so one knows fairly quickly that something is wrong.
3. **Re-use**: How easy is it to ensure that the semantics of a bytecode instruction are expressed in as few places as possible?
    - Other parts of the VM can simply call snippet code as normal Java methods.
4. **Portability**: How much needs to be changed when porting the VM to a new platform?
    - The snippets include no machine code or even any compiler-specific code. Any platform dependent code in a snippet is expressed as Java code that tests a compile-time constant platform configuration value. As long as the compiler implements the protocol required for bootstrapping the snippets, there should no need to modify any other parts of a replacement compiler.
5. **Syntactic correctness**: How easy it is to verify that snippets are syntactically correct?
    - As easy as having the Java source code compiler successfully compile the snippet source code!
6. **Optimization potential**: How much do snippets enhance or inhibit optimization potential in a compiler?
    - Snippet IR is designed to be woven into the IR of a method before optimization. This means all snippet IR is subject to complete optimization in the context of the method being compiled. So, in theory, a compiler based on snippets should allow maximum optimization of the code paths that implement the runtime/compiler interface. However, it also means that the quality of code generated for these code paths is at the mercy of the compiler. [STRIKEOUT:Due to the sub-optimal CPS compiler in Maxine, the code derived from snippets is far from optimal.](outdated)
7. **Compiler design**: How much do snippets complicate or simplify a compiler's design?
    - This point can only be accurately addressed in the same way proposed for the **Performance** question. Only then can one accurately comment on the architectural impact of making a compiler snippet aware.
8. **Locality**: How easy is it to find and navigate the code related to a single snippet?
    - This is one of the weaker aspects of snippets as they are currently implemented in the Maxine VM. The source code for the snippets is distributed amongst many classes, one class per snippet. The properties of some snippets are encoded in the snippet class hierarchy. For example, all snippets whose optimized IR must not include any calls (except to builtins) must

subclass the `BuiltinsSnippet` class while those that cannot be folded must subclass the `NonfoldableSnippet` class. All such compilation-properties of snippets should really be associated with the snippet entry point, possibly as elements of the `@SNIPPET` annotation. In addition, the way in which snippets are discovered and registered with the compiler is more complicated than it should be, relying on class initialization. [STRIKEOUT:Most of these issues however, are simply code engineering issues that are relatively easy to remedy, especially if modeled after the way in which XIR snippets are organized.](outdated)

9. **Code Layout**: What's the granularity of control over how the generated code is organized?
    - The code path for snippets is either inlined or involves a runtime call. [STRIKEOUT:Like XIR, one would ideally like to be able to express fast inline path, out-of-line but in method path, global stub and runtime call paths.](outdated) With some careful thought, modifying or augmenting the `@INLINE` annotation may enable such code-layout to be expressed.

7.18 Stack Walking in the Maxine VM

This page documents the stack walking mechanism in the Maxine VM. It details the uses of stack walking as well as special considerations that need to be taken into account by the stack walking mechanism.

A stack walk is a traversal through the frames of the stack going from callee frames to caller frames. That is, the walk through the frames is performed in the reverse order in which the frames were pushed on the stack. This reflects the fact that callee frames always know the execution context of their caller's frame but the inverse is not true. The execution context of interest for a given frame is the value of the instruction position, stack pointer and frame pointer in the frame. Note that the exact meaning of a caller's instruction pointer depends on the underlying platform. For example, on x86 platforms, the `CALL` instruction pushes the address of the instruction following the `CALL` instruction to the stack. On SPARC systems, the `CALL` instruction saves the address of the `CALL` instruction in the link register, `%i7`. This distinction is critical when searching the stack for exception handlers (more on this below).

A stack walk is always performed for a specific reason or purpose which are detailed below.

7.18.1 Stack walk purpose 1: Exception handling

When an exception is thrown in Java, the stack of the current thread is searched for an appropriate exception handler. For each target method corresponding to a frame on the stack, a search is performed for an exception handler based on the current instruction position in the target method and the type of the exception. Searching a target method based on these parameters takes into account how the exception handler information is encoded.

Once an exception handler is found, the stack frame walker adjusts the current execution context to affect a transfer of control to the exception handler. This usually just involves updating the current stack, frame and instruction pointers. However, there are two special cases where a little more has to be done when restoring the state before executing the exception handler:

- **Implicit Exceptions:** An implicit exception is one caused by a trap. In Maxine, the implicit exceptions resulting from traps are null pointer exceptions, divide-by-zero exceptions and stack overflow errors. When an implicit exception occurs in a frame that has an exception handler for it, then the register state at the exception point must be restored before jumping to the exception handler. This allows a register allocator to operate on the assumption there is a direct control flow edge from any instruction that may cause an implicit exception to a local exception handler covering that instruction.

- **Stack Overflow:** Stack overflow detection is implemented by using protecting a guard page near the end of the stack. When a stack overflow error occurs, the guard page was unprotected by the low-level trap handler. Before jumping to the exception handler, the guard page must be reprotected.

7.18.2 Stack walk purpose 2: Stack reference map preparation

The garbage collector needs information about which stack slots and registers contain object references; this information is stored in the stack reference map in thread-local memory. When preparing a reference map, three cases can be distinguished:

- Top-most method. This method provides information about its stack and its registers.
- Caller saved method. This method provides information about its stack. When the method is at a position where it called a callee saved method, it must provide a reference map for its registers too.
- Callee-saved method. All references of this method lie on the stack. However, if they are references or not depends on the registers of the caller method. Therefore this method must provide a reference map that contains a bit for each register that is saved on the stack.

Note that this model imposes the following constraints:

1. A callee-saved method cannot call another callee-saved method.
2. A callee-saved method cannot be on the top of the stack.

One other important constraint on stack walking for reference map preparation is that it must never perform any heap-allocation.

7.18.3 Stack walk purpose 3: Stack inspection

There are a number of subsystems in the VM that needs to inspect the frames on a stack. These are lumped together under the purpose of stack inspection and perform one of the two following inspections supported by the stack frame walker:

1. **Low Level Inspection:** The stack frame walker allocates no memory when performing this kind of inspection. The client is notified (via the visitor pattern) of the following information for each stack frame traversed:
    - Stack pointer
    - Frame pointer
    - Instruction pointer
    - Target method (optional) This stack walk is used with different visitors for exception handling, reference map preparation, the *inspector*, and deoptimization.
2. **High Level Inspection:** The stack frame walker returns a list of method objects denoting the methods for each frame on the stack.

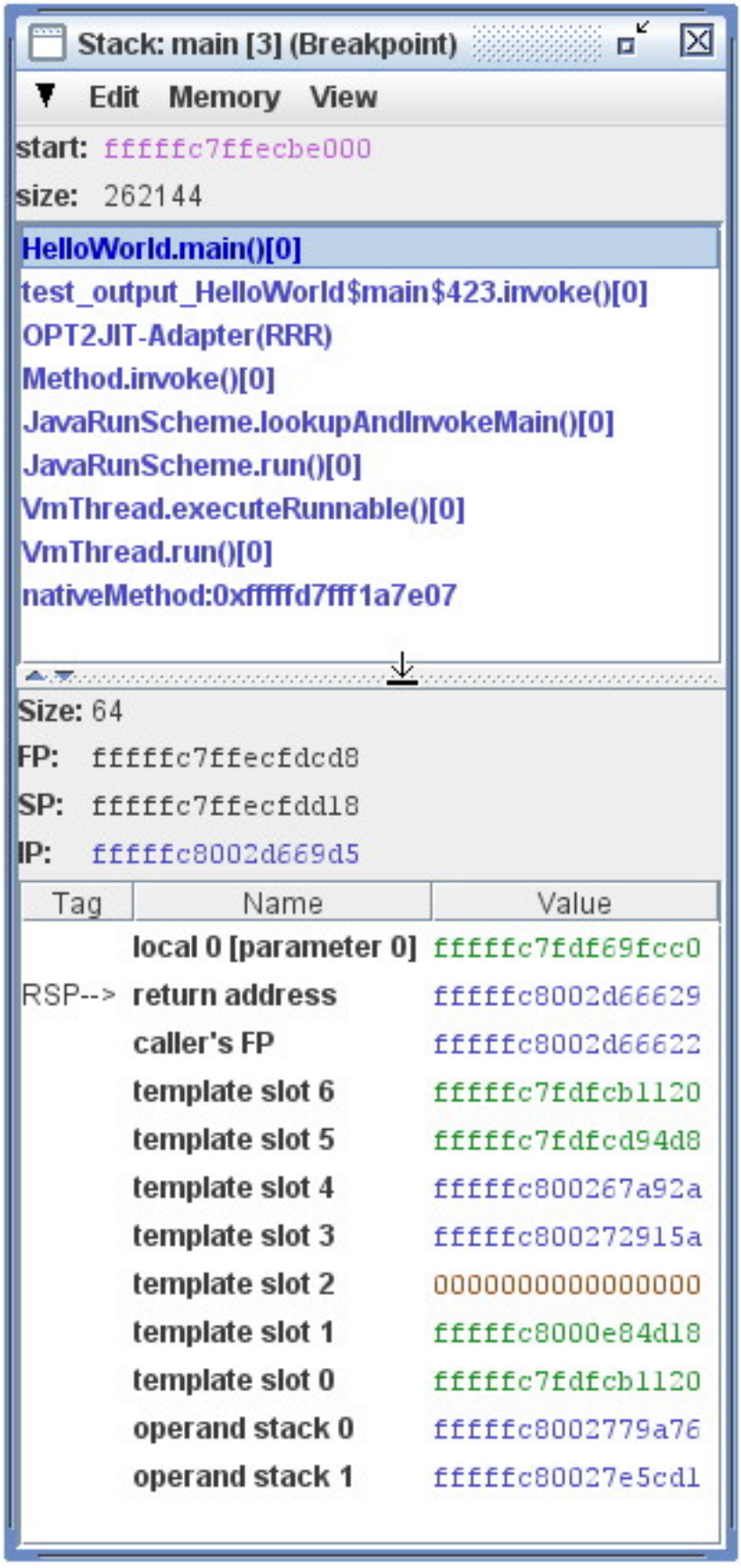
The subsystems using a stack inspection are detailed below.

Creating a stack trace for an exception

When walking the stack for creating a stack trace, the stack frame walker must distinguish between frames that should be included in the stack trace (application visible methods) and those that should not. An object of type java.lang.StackTraceElement is created for each method included in the stack trace.

Inspector

The Maxine Inspector wants to get a detailed view of all stack frames for display in the Stack Inspector view. For Java methods at stop positions or JIT methods, the JVM bytecode level local variables, operand stack and monitor state is of interest. These are the same data structures that deoptimization, on-stack-replacement, or a Java interpreter would work with.


Stack: main [3] (Breakpoint)
Edit Memory View
start: ffffffc7ffecbe000
size: 262144
HelloWorld.main()[0]
test_output_HelloWorld$main$423.invoke()[0]
OPT2JIT-Adapter(RRR)
Method.invoke()[0]
JavaRunScheme.lookupAndInvokeMain()[0]
JavaRunScheme.run()[0]
VmThread.executeRunnable()[0]
VmThread.run()[0]
nativeMethod:0xfffffd7fff1a7e07
Size: 64
FP: ffffffc7ffecfdcd8
SP: ffffffc7ffecfdd18
IP: ffffffc8002d669d5
Tag Name Value
local 0 [parameter 0] ffffffc7fdf69fcc0
RSP--> return address ffffffc8002d66629
caller's FP ffffffc8002d66622
template slot 6 ffffffc7fdfcb1120
template slot 5 ffffffc7fdfcd94d8
template slot 4 ffffffc800267a92a
template slot 3 ffffffc800272915a
template slot 2 0000000000000000
template slot 1 ffffffc8000e84d18
template slot 0 ffffffc7fdfcb1120
operand stack 0 ffffffc8002779a76
operand stack 1 ffffffc80027e5cd1

JDK

The following JDK methods require inspecting the stack. They all only operate on methods that have an associated class method actor.

- `java.security.AccessController.getProtectionDomains()`
- `sun.reflect.Reflection.getCallerMethod(int)`
- `JVM_LatestUserDefinedLoader` (defined in `jvm.h`)

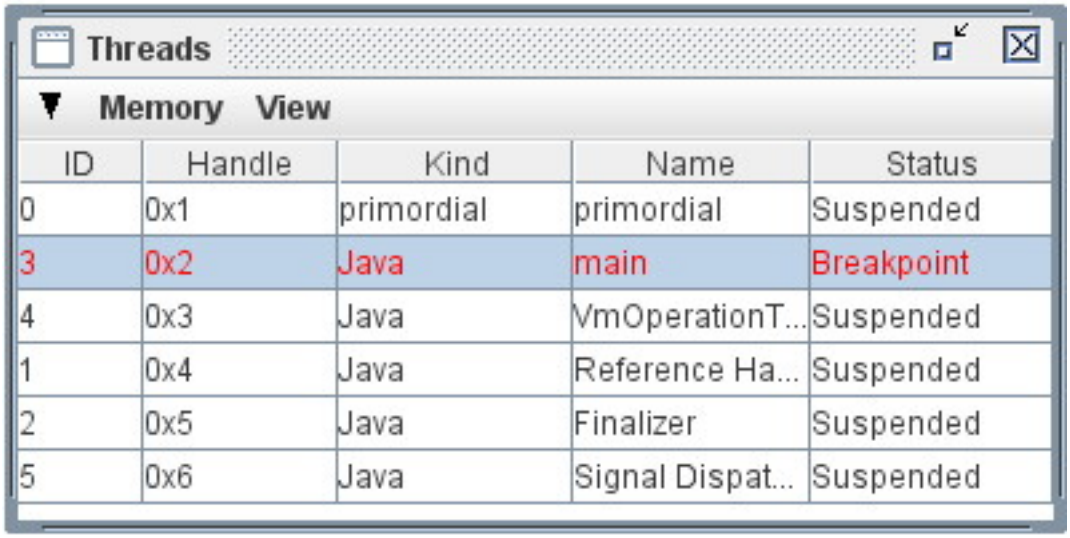
Deoptimization

In order to deoptimize a certain method that is not the top most method, the stack frame walker needs to patch the return address of a method to a different location (that performs the deoptimization and continues execution in the interpreter). Note that also when this method is selected to perform a catch for an exception, deoptimization must be performed.

7.19 Threads in the Maxine VM

Maxine threads are implemented using native threads of the underlying operating system, in contrast to the "green threads" approach. Each thread is represented in the VM as an instance of class `com.sun.max.vm.thread.VmThread`.

The *Threads Inspector View* of the *Maxine Inspector* displays information about all currently existing threads in the VM.



7.19.1 The main thread

The main thread running in a new Maxine VM process executes the preliminary steps taken during VM startup, up until the creation and running of the main Java thread. The C main function runs on this thread.

The main thread (which appears initially in the Inspector as an "unnamed native" thread), eventually becomes the thread on which the Java main thread runs.

7.19.2 Thread local memory

The VM allocates an area of memory for each thread known as the thread locals block, separate from the thread's stack. This area includes copies of thread local variables for VM internal use, and a stack reference map, among others.

Thread local variables

The VM provides an extensible mechanism for allocating per-thread variables, often referred to simply as *thread locals*. This mechanism is used by VM internals; it should not be confused with language-level Thread local storage, which is supported in Java by the `class java.lang.ThreadLocal`.

Each instance of class `com.sun.max.vm.thread.VmThreadLocal` automatically creates a per-thread word-length variable with a well-defined read/write protocol. That protocol depends on each variable's nature (an instance of the enum `com.sun.max.vm.thread.VmThreadLocal.Nature`), which is fixed at creation. Many thread local variables are created as static members of the defining class `VmThreadLocal`. Thread locals are also defined by other parts of the VM as needed.

Each thread local has a fixed name, assigned at creation, by which it can be referenced within the VM, along with a `boolean` that specifies whether the variable will hold references. Each variable is also assigned a description, a terse human-readable description of the variable's purpose that is accessed by the Maxine Inspector and made available to users in the Thread Locals View.


Thread Local Variables: main [3] (Breakpoint)
Memory View
TRIGGERED ENABLED DISABLED
start: 00000000041bff8 end: 00000000041c110 size: 280
T... Pos. Field Value
+0 SAFEPOINT_LATCH void
+8 SAFEPOINTS_ENABLED_THREAD_LO... 00000000041c118
+16 SAFEPOINTS_DISABLED_THREAD_LO... 00000000041c238
+24 SAFEPOINTS_TRIGGERED_THREAD_L... 00000000041bff8
+32 NATIVE_THREAD_LOCALS 00000000041c358
+40 FORWARD_LINK 0000000000000000
+48 BACKWARD_LINK 0000000000425118
+56 VM_OPERATION null
+64 EXCEPTION_OBJECT null
+72 ID 0000000000000003
+80 VM_THREAD <29833>VmThread{mai...
+88 JNI_ENV ffffffd7fff1c2010
+96 LAST_JAVA_FRAME_ANCHOR 0000000000000000
+1... MUTATOR_STATE 0000000000000000
+1... FROZEN 0000000000000000
+1... TRAP_NUMBER 0000000000000000
+1... TRAP_INSTRUCTION_POINTER 0000000000000000
+1... TRAP_FAULT_ADDRESS 0000000000000000
+1... TRAP_LATCH_REGISTER 0000000000000000
+1... HIGHEST_STACK_SLOT_ADDRESS ffffffc7ffecfe000
+1... LOWEST_STACK_SLOT_ADDRESS ffffffc7ffecbf000
+1... LOWEST_ACTIVE_STACK_SLOT_ADD... 0000000000000000
+1... STACK_REFERENCE_MAP 00000000041c3a8
+1... STACK_REFERENCE_SIZE 0000000000001008
+1... IMMORTAL_ALLOCATION_ENABLED 0000000000000000
+2... INTERPRETED_METHOD null
+2... NATIVE_CALL_STACK_SIZE 0000000000000000
+2... TLAB_TOP 0000000000000000
+2... TLAB_MARK 0000000000000000
+2... TLAB_TOP_TMP 0000000000000000

Thread locals area (TLA)

During boot image generation, a contiguous block of memory known as the thread locals area (TLA) is defined to contain a word for each thread local variable, and each variable is assigned an offset into the TLA. The first location in each TLA is reserved for the "safepoint latch".

Three TLAs (with identical layout) are defined for each thread, one corresponding to each of the VM's three safepoint states: Enabled, Disabled, and Triggered. This design permits efficient implementation of both safepoints and thread locals access. Since a dedicated register (R14 on x64) points to the TLA for the current safepoint state, this means both safepoints and thread local variable access can be performed with one or two loads and, more importantly, without control flow operations.

The base location of the three TLAs is recorded in the thread locals named `ETLA`, `DTLA`, and `TTLA` respectively.

Stack reference map

Each thread has an associated stack reference map, a data structure that identifies the thread's stack locations that hold references to the heap. A `StackReferenceMapPreparer` prepares reference maps on demand (using a stack walk, see "Stack reference map preparation"), just after a thread has entered the frozen state (and is therefore at a safepoint) because of a GC operation.

The map uses one bit per word on the stack so it is about 3% of the stack size on a 32-bit system and about 1.5% on a 64-bit system.

See class `com.sun.max.vm.stack.StackReferenceMapPreparer`.

Thread locals block (TLB)

Per-thread VM storage is in a separate thread locals block (TLB) that is allocated in native code and freed by the native thread library mechanism for destructing thread specific keys (e.g. the second argument of `pthreadkeycreate(3)`). This permits attaching native threads via JNI, where there is no way to carve out a piece of the stack for the VM.

The TLB includes not only three Thread locals areas (TLAs) but also other thread-specific data such as the stack reference map. The layout of the TLB is shown in the following diagram, copied from the JavaDoc comments for class `com.sun.max.vm.thread.VmThreadLocal`:

```
(low addresses)

  page aligned --> +-----------------------------------------------+
                   | X X X          unmapped page          X X X |
                   | X X X                                 X X X |
  page aligned --> +-----------------------------------------------+
                   |                TLA (triggered)                |
                   +-----------------------------------------------+
                   |                TLA (enabled)                  |
                   +-----------------------------------------------+
                   |                TLA (disabled)                 |
                   +-----------------------------------------------+
                   |           NativeThreadLocalsStruct            |
                   +-----------------------------------------------+
                   |                                               |
```

(continues on next page)

(continued from previous page)

```
                    |                reference map                |
                    |                                             |
                    +---------------------------------------------+

(high addresses)
```

You can use the `-XX:+TraceThreads` VM option to see the layout of the stack and TLB for each thread as it starts.

```
Initialization completed for thread[id=3, name="main", native␣
↪id=0x100096000]:
Stack layout:

 +--------- 0x100096000  [+262144]
 |
 | OS thread specific data and native frames [720 bytes, 0.274658%]
 |
 +--------- 0x100095d30 [+261424]
 |
 | Frame of Java methods, native stubs and native functions [257328 bytes,␣
↪98.162842%]
 |
 +--------- 0x100057000 [+4096]
 |
 | Stack yellow zone [4096 bytes, 1.562500%]
 |
 +--------- 0x100056000 [+0]
 |
 | Stack red zone [4096 bytes, 1.562500%]
 |
 +--------- 0x100055000 [-4096]

Thread locals block layout:
 +--------- 0x10083a380  [+9088]
 |
 | reference map [4104 bytes, 45.158451%]
 |
 +--------- 0x100839378 [+4984]
 |
 | native thread locals [80 bytes, 0.880282%]
 |
 +--------- 0x100839328 [+4904]
 |
 | safepoints-disabled thread locals area [272 bytes, 2.992958%]
 |
 +--------- 0x100839218 [+4632]
 |
 | safepoints-enabled thread locals area [272 bytes, 2.992958%]
 |
 +--------- 0x100839108 [+4360]
 |
 | safepoints-triggered thread locals area [272 bytes, 2.992958%]
 |
 +--------- 0x100838ff8 [+4088]
 |
 | unmapped page [4088 bytes, 44.982395%]
```

(continues on next page)

(continued from previous page)

```
 |
 +--------- 0x100838000 [+0]
```

7.19.3 Safepoints

A safepoint is a special instruction in compiled VM code, at a location where a thread can be frozen with guaranteed consistency between the thread's stack and the heap, which is required for safe garbage collection. Maxine compilers insert safepoints in branches, goto, and switch statements.

A safepoint incurs very low overhead in normal operation, but causes a trap when triggered in the thread; this typically happens when the VM is preparing for garbage collection.

Abstract class `com.sun.max.vm.runtime.Safepoint` is specialized by subclasses with platform-specific details of safepoint implementation.

7.19.4 Stack overflow detection

To implement stack overflow detection (which can result in raising a `StackOverflowError`), Maxine places guard pages at the limit of the stack. More precisely, Maxine uses OS page protection facilities (see `mprotect(2)`) to make a couple of pages at the end of the stack non-readable and non-writable. This enables stack overflow detection to be performed by a single instruction in the prologue of a method. Mostly this instruction is effectively a no-op (i.e. has no side-effect visible to the program). For example, the following stack banging instruction is used in Maxine on AMD64 to load a value from a fixed (negative) offset from the stack pointer:

```
mov r11, [rsp - 12288] # load from 3 pages below %rsp
```

To understand how this may cause a trap, consider the following layout of a thread's stack in Maxine:

```
High addresses

                        +---------------------------------------------+
                        |          OS thread specific data            |
                        |           and native frames                 |
                        +---------------------------------------------+
                        |                                             |
                        |           Frames of Java methods,           |
     stack pointer -->  |              native stubs, and              |
                        |              native functions               |
                        |                                             |
                        +---------------------------------------------+
                        | X X X     Stack overflow detection    X X X |
                        | X X X          (yellow zone)          X X X |
      page aligned -->  +---------------------------------------------+
                        | X X X     Stack overflow detection    X X X |
                        | X X X           (red zone)            X X X |
      page aligned -->  +---------------------------------------------+

Low addresses
```

If the value of `%rsp - 12288` lies within the yellow zone, then a `SIGSEGV` signal will be raised. Maxine's VM signal handler will then test whether or not the faulting address lies within the yellow zone. If it does, then the protection bits of the yellow zone are modified such that further

reads/writes to this page will not cause a trap. This should allow the code that allocates and raises a `StackOverflowError` to execute without causing stack overflow itself. Just before returning to the exception handler, the yellow zone is re-guarded. The red zone exists to detect the situation where the stack overflow raising code uses too much stack. This is a fatal VM error. It's also a fatal VM error if stack overflow occurs when execution is in native code (called via JNI).

7.19.5 Thread local allocation buffer (TLAB)

A thread local allocation buffer (TLAB) is a portion of heap storage reserved for allocation by a single thread. This allows heap allocation without synchronization, typically via a simple pointer increment. Fast access to the thread's TLAB is provided via thread local variables stored in the *Thread locals area (TLA)*. Most object allocation goes via the TLAB of the thread requesting the allocation first.

When a thread has exhausted its TLAB, it is refilled with a new one. TLAB refill decisions are driven by a `TLABRefillPolicy`.

Because the logic of TLAB management and allocation is common to all implementations of `HeapScheme`, it is factored in the adaptor class `com.sun.max.vm.heap.HeapSchemeWithTLAB`.

Aspects of TLAB management that depend on `HeapScheme`'s details are delegated to the concrete implementations. These includes: handling requests that overflow the TLAB's current free space, refilling the TLAB with new heap space, actions to be taken on TLAB refill, making the TLAB parseable at GC safepoint, or the choice of TLAB refill policy.

7.20 Type-based Logging

Type-based logging is the preferred way to log, and optionally trace, values of interest to aid in the development and debugging of Maxine. The term type-based indicates that the logging methods use statically typed values, rather than doing up-front conversion to String values as in string-based logging.

The standard method for debugging Maxine is interactively with the Maxine Inspector. However, there are times when pure interactive debugging is inadequate, for example, in complex multi-threaded situations. To address this Maxine has, historically, used tracing calls, embedded in the VM source code and using the Log class, that output specific data to either the standard output or a file. This approach has some drawbacks:

- Although largely string based, the tracing calls must follow strict rules regarding GC interaction and multi-threading.
- The source code can become obfuscated by the tracing code.
- There format of the tracing output is fixed by the tracing calls.
- There is no connection to the Maxine Inspector.

The framework provided by `VMLogger` attempts to address these drawbacks in the following ways:

- Replace the tracing generation calls with more abstract calls to methods in `VMLogger`.
- Handle multi-threading, GC issues automatically.
- Integrate the VMLogger data with the Maxine Inspector.
- Provide optional custom tracing in the style of Log but driven from the log.

Note, the name log is overloaded. The existing `Log` class is not a log in the sense defined by `VMLogger`, rather it is a mechanism for printing strings, scalars and some object types, e.g., threads, to an output stream. In other words it is message oriented, similar to the platform logging framework. `VMLogger` is more "type" oriented and is targeted towards in-memory log storage, with log inspection handled by the *Maxine Inspector*. By storing object values directly in the log, rather than a string encoding, the Inspector mechanisms for drilling down into the fields of an object can be exploited. In the following, we refer to string-based logging as tracing.

The expectation is that each component, or module, of the VM has one or more associated loggers. Loggers are identified by a short name and a longer description. A given logger is disabled by default but can be enabled with a command line option at VM startup. **Note:** A Logger can be enabled in hosted mode if that is appropriate for the VM component. All logger state is reset when the target VM starts, so host settings do not persist.

It is also expected is that most loggers will be implemented using the automatic generation features of `VMLoggerGenerator` and not be hand-written, except as regards custom tracing support. See the section below entitled *Automatic Generation*.

`VMLogger` does not define the implementation of the log storage. This is handled by `VMLog`, which is an abstract class that is capable of several implementations, with various tradeoffs regarding space requirements and performance.

7.20.1 VMLogger

A `VMLogger` defines a set of operations, cardinality `N` each identified by an `int` code in the range `[0 .. N-1]`. A series of log methods are provided, that take the operation code and a varying number of `Word` arguments (up to `VMLog.Record.MAX_ARGS`). Each log operation creates a log record that is stored in a circular buffer, the size of which is determined when the VM image is built. The thread (id) generating the log record is automatically recorded.

In order to connect the operation code with a `String` value that can be used to identify the operation, e.g. for tracing, VM startup options, etc., a logger should provide an overriding implementation of `VMLogger.operationName(int)` that returns a descriptive name for the operation.

Enabling Logging

Logging is enabled on a per logger basis through the use of a standard `-XX:+LogMMM` option derived from the logger name, in this case `MMM`. Tracing to the Log stream is also available through `-XX:+TraceMMM`. A default tracing implementation is provided, although this can be overridden by a given logger. Enabling tracing also enables logging, as the trace is driven from the log. **Note:** It is not possible to check the options until the VM startup has reached a certain point. In order not to lose logging in the early phases, logging, but not tracing, is always enabled on VM startup.

Fine control over which operations are logged (and therefore traced) is provided by the `-XX:LogMMMInclude=pattern` and `-XX:LogMMMExclude=pattern` options. The pattern is a regular expression in the syntax expected by `Pattern` and refers to the operation names returned by `VMLogger.operationName`. By default all operations are logged. However, if the include option is set, only those operations that match the pattern are logged. In either case, if the exclude option is provided, the set is reduced by those operations that match the exclude pattern.

The management of log records is handled in a separate class; a subclass of `VMLog`. A instance requests a record that can store a given number of arguments from the singleton `VMLog.vmLog` instance and then records the values. The format of the log record is opaque to allow a variety of implementations.

Performance Concerns

In simple use logging affects performance even when disabled because the disabled check happens inside the `VMLogger` log methods, so the cost of the argument evaluation and method call is always paid when used in the straightforward manner, e.g.: `logger.log(op, arg1, arg2);`

It is recommended that all log calls be guarded as follows:

```
if (logger.enabled()) {
    logger.log(op, arg1, arg2);
}
```

The enabled method is always inlined.

Note: The guard can be a more complex condition. However, it is important not to use disjunctive conditions that could result in a value of true for the guard when `logger.enabled()` would return false, E.g.,

```
if {a || b} {
    logger.log(op, arg1, arg2);
}
```

Conjunctive conditions can be useful. For example, say we wanted to suppress logging until a counter reaches a certain value:

```
if (logger.enabled() && count >= value) {
    logger.log(op, arg1, arg2);
}
```

Dependent Loggers

It is possible to have one logger override the default settings for other loggers. E.g., say we have loggers `A` and `B`, but we want a way to turn both loggers on with a single overriding option. The way to do this is to create a logger, say `ALL`, typically with no operations, that forces `A` and `B` into the enabled state if, and only if, it is itself enabled. This can be achieved by overriding `VMLogger.checkOptions()` for the `ALL` logger, and calling the `VMLogger.forceDependentLoggerState` method. See `Heap.gcAllLogger` for an example of this.

It is also possible for a logger, say `C`, to inherit the settings of another logger, say `ALL`, again by forcing `ALL` to check its options from within `C`'s checkOptions and then use `ALL`'s values to set `C`'s settings. This is appropriate when `ALL` cannot know about `C` for abstraction reasons. See `VMLogger.checkDominantLoggerOptions`.

Note: The order in which loggers have their options checked by the normal VM startup is unspecified. Hence, a logger must always force the checking of a dependent logger's options before accessing its state.

Logging (for all loggers) may be enabled/disabled for a given thread, which can be useful to avoid unwanted recursion in low-level code, see `VMLog.setThreadState`.

Automatic Generation

The standard type-safe way to log a collection of heteregenously typed values would be to first define a class containing fields that correspond to the values, then acquire an instance of such a class, store the

values in the fields and then save the instance in the log. Note that this generally involves allocation; at best it involves acquiring a pre-allocated instance in some way. It also necessarily involves a level of indirection in the log buffer itself, as the buffer is constrained to be a container of reference values. Since VM logging is a low level mechanism that must function in parts of the VM where allocation is impossible, for example, during garbage collection, the standard approach is not appropriate. It is also important to minimize the storage overhead for log records and the performance overhead of logging the data. Therefore, a less type safe approach is adopted, that is partly mitigated by automatic generation of logger code at VM image build time.

The automatic generation, see `VMLoggerGenerator`, is driven from an interface defining the logger operations that is tagged with the `VMLoggerInterface` annotation. Since this is only used during image generation the interface should also be tagged with `HOSTED_ONLY`. The logging operations are defined as methods in the interface. In order to preserve the parameter names in the generated code, each parameter should also be annotated with `VMLogParam`, e.g.:

```
@HOSTED_ONLY
@VMLoggerInterface
private interface ExampleLoggerInterface {
  void foo(
      @VMLogParam(name = "classActor") ClassActor classActor,
      @VMLogParam(name = "base") Pointer base);

  void bar(
      @VMLogParam(name = "count") SomeClass someClass, int count);
}
```

The logger class should contain the comment pair:

```
// START GENERATED CODE
// END GENERATED CODE
```

somewhere in the source, typically at the end of the class. When `VMLoggerGenerator` is executed it scans all VM classes for interfaces annotated with `VMLoggerInterface` and then generates an abstract class containing the log methods, abstract method definitions for the associated trace methods, and an implementation of the `VMLogger.trace` method that decodes the operation and invokes the appropriate trace method.

The developer then defines the concrete implementation class that inherits from the automatically generated class and, if required implements the trace methods, e.g, from the `ExampleLoggerOwner` class:

```
public static final class ExampleLogger extends ExampleLoggerAuto {
     ExampleLogger() {
        super("Example", "an example logger.");
     }

    @Override
    protected void traceFoo(ClassActor classActor, Pointer base) {
        Log.print("Class "); Log.print(classActor.name.string);
        Log.print(", base:"); Log.println(base);
    }

    @Override
    protected void traceBar(SomeClass someClass, int count) {
      // SomeClass specific tracing
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Note that if an argument name is not identified with `VMLogParam` it will be defined as `argN`, where `N` is the argument index.

`VMLogger` has built-in support for several standard reference types, that have alternate representations as scalar values, such as `ClassActor`. As a general principle, reference types without an alternate, unique, scalar representation should be avoided as log method arguments. However, this is sometimes difficult or inconvenient, so it is possible to store references types. These should be passed using `VMLogger.objectArg` and retrieved using `VMLogger.toObject`. This is automatically handled by the generator. **Note:** Storing reference types in the log makes them reachable until such time as they are overwritten. It is assumed that `Enum` types are always stored using their ordinal value. The generator creates the appropriate conversions methods. It assumes that the enum declares the following field:

```
public static final EnumType[] VALUES = values();
```

Tracing

When the tracing option for a logger is enabled, `VMLogger.doTrace` is invoked immediately after the log record is created. After checking that calls to the Log class are possible, `Log.lock` is called, then `VMLogger.trace` is called, followed by `Log.unlock`.

A default implementation of `VMLogger` is provided that calls methods in the `Log` class to print the logger name, thread name and arguments. There are two ways to customize the output. The first is to override the `VMLogger.logArg(int, Word)` method to customize the output of a particular argument - the default action is to print the value as a hex number. The second is to override `VMLogger.trace` and do full customization. **Note:** Although the log is locked automatically and safepoints are disabled, custom tracing must still take care not to invoke object allocation. In particular, string concatenation and formatting should not be used.

7.20.2 Inspector Integration

The *Inspector* is generally able to display the log arguments appropriately, by using reflection to discover the types of the arguments.

Two additional mechanisms are available for Inspector customization. The first is an override to generate a custom String representation of a log argument:

```
@HOSTED_ONLY
public String inspectedArgValue(int op, int argNum, Word argValue);
```

If this method is defined for a given logger then the Inspector will call it for the given operation and argument and, if it returns a non-null value, use the result.

The second is an override for a logger-defined argument value class:

```
@HOSTED_ONLY
public static String inspectedValue(Word argValue);
```

If this method is defined for the class and no standard customization is available, it will be called and, if the result is non-null it will be used.

7.20.3 VMLog

`VMLog` maintains the global table of `VMLogger` instances, and provides the log storage implementation and support for interacting with the garbage collector. The actual log storage implementation is specified by abstract methods and a particular implementation is chosen at VM image build time. The default implementation is `VMLogNativeThreadVariable` which stores log records in a per-thread native buffer. The other implementation that is provided with Maxine is `VMLogArrayFixed`, which can be enabled by setting the `max.vmlog.class` system property to `java.fix.VMLogArrayFixed`. This is an all-Java implementation that uses a global buffer comprising an array of fixed length `VMLog.Record` instances. It should be used as a check if there is a suspicion that the default implementation is manifesting a bug.

7.20.4 VMLog Flushing

By default, older log records are overwritten when the circular buffer wraps around. In normal use this is not a problem, as the Inspector maintains all the log records in its own non-circular buffer. However, in exceptional circumstances, for example when not running the Inspector, it may be convenient to flush the log, say on a VM crash, rather than tracing every log operation. This can be enabled with the `-XX:VMLogFlush=setting` VM option. The value of setting should be a comma separated string contains one of the following:

- `crash`: flush the log on a VM crash
- `exit`: flush the log on normal VM exit
- `full`: flush the log whenever it becomes full (i.e., is about to overwrite old records)
- `raw`: output the log records as uninterpreted, raw, bits.
- `trace`: output the log records using the `VMLogger.trace` method

The default output mode is `raw`, which is robust, but requires offline interpretation. `Trace` mode may be unstable after a VM crash as it may provoke a recursive crash.

Note that flushing the log when full, using `trace` mode output, is essentially equivalent to setting the associated trace options, except for that the data might be "stale" by delaying the interpretation until the log is flushed.

The *Maxine Inspector* can interpret a file of `VMLog` records using `mx view -vmlog=file`. The simplest way to create the file is to redirect the log output to a file by setting export `MAXINE_LOG_FILE=maxine.log` before running the VM, and then copying the file. The last step is important because the Inspector will overwrite the log file when it executes (meta-circularity!).

Automatically generated from com.sun.max.vm.log.package-info

7.21 Virtual Machine Level Analysis

Maxine contains an experimental extension to support analysis of code executing on the virtual machine (VM). Since the VM is itself written in Java, the analysis is applicable, in principle, to the VM itself, the

platform (JDK) and the application.

The analysis is implemented primarily by advising the execution of the bytecodes. This is similar to systems like AspectJ except that the advice is applied to the bytecodes and not to language-level constructs. However, since the translation of language-level constructs to bytecodes is well-defined, language-level advising could, in principle, be added as a separate layer. Certain other VM runtime operations, for example, garbage collection, thread start/end can also be advised.

The Virtual Machine Level Analysis (VMA) project is currently implemented as an extension to Maxine in the `com.oracle.max.vm.ext.vma` and `com.oracle.max.vma.tools` projects. The implementation exploits the flexibility inherent in the Maxine VM by defining custom versions of several existing Maxine schemes, and building a custom VM image.

Currently bytecode advising is limited to code generated by the (template) JIT compiler, although it is expected to be added to the optimizing compiler in due course.

VMA shares some similarity with aspects of the JVMTI API, most notably the method entry/exit and field access/watch capabilities and some of the runtime advice, e.g. thread start/end. Eventually, the redundant features may be removed and the two implementations merged. In the interim, some analyses can benefit from both systems, using Maxine's Java JVMTI interface JJVMTI.

There is the beginnings of VMA for Graal in the `com.oracle.max.vm.ext.vma` project. However, it should be considered experimental.

7.21.1 Architecture

To facilitate experimentation the VMA architecture is highly flexible and configurable. The system contains four basic components:

1. a custom version of the T1X JIT compiler (VMAT1X) that uses custom templates for adding advice at bytecode translation time.
2. the VMA runtime which invokes methods in an advice handling class that can be specified either at VM image build time or loaded dynamically as a VM extension.
3. a store interface that supports the persistent storage of advice data for offline analysis, with an implementation that can
4. be specified at runtime.
5. a tool that can execute queries against the data in the persistent store.

Note that advising is disabled while in the scope of a handler execution, to prevent recursive entry from instrumented code that may be shared by the handler.

7.21.2 Building a VMA-enabled image

```
mx image @vma-t1x
```

This VM image includes the custom VMA schemes but does not specify an advice handler which must, therefore be loaded dynamically using the VM extension mechanism. By default advising is enabled, and if no handler or built into the VM Image or loaded as a VM extension, the VM will abort. To disable advising set `-XX:-VMA` option.

7.21.3 Running a VMA-enabled image

A VMA-enabled image can be run in the usual way with the `max vm` command. When advising is enabled (the default), the VMAT1X compiler is used for compiling dynamically loaded classes. This compiler will use the advice-enabled T1X templates. By default all methods in dynamically loaded classes are processed by VMAT1X. However, this can be controlled more precisely with the `-XX:VMAMI` and `-XX:VMAMX` options. Both options take values that are regular expressions in the format expected by `java,util.regex.Pattern`. The `-XX:VMAMI` option specifies methods to include and `-XX:VMAMX` specifies methods to exclude. If `-XX:VMAMI` is unset, all methods are candidates for processing with VMAT1X, otherwise only those methods that match the pattern. In either case, if the `-XX:VMAMX` is also set, it excludes any method in the candidate set that matches its pattern. The syntax of a method pattern is `qualified_classname#methodnameunqualified_param_class_1, unqualified_param_class_2, ...`. Note that the parameter list braces must be escaped, as must the square braces in any array class specifier. The wildcard specifier `.` can be used to match all method names and/or signatures.

The exact set of bytecodes that have advice applied during compilation can also be controlled with options. By default, all bytecodes are advised but, depending on the desired analysis, a subset is usually more appropriate. While it is possible to be quite specific about exactly which bytecodes will be advised, there are some predefined configurations set via the `VMAConfig` option that are easier to use, e.g.:

- `entry`: advises after method entry
- `entryexit`: advises after method entry and before return, including return by thrown exception
- `monitor`: advises the acquisition and release of monitors.
- `read`: advises any access to an object's state. An access is defined as any read of an object's fields or its metadata or, if an array, it's elements.
- `write`: advises any write to an object's fields or, if an array, any of its elements.
- `objectaccess`: advises the creation of objects and all access sites.
- `objectuse`: advises the creation of objects and all usage sites. A use is defined as any load or store of the object reference, or any access.

The `VMAConfig` options accepts a list of these configurations and aggregates them into a single set of the appropriate bytecodes to be advised.

The specific bytecode advice controls are `-XX:VMABI` and `-XX:VMABX` and behave very similarly to the class controls. Their argument is a regular expression matching bytecodes to be included or excluded, respectively. To provide control over enabling before/after advice a bytecode may be suffixed by `:A`, `:B` or `:AB`. Note that while all the bytecodes in the standard set defined by the JVM can be individually controlled, the `VMAdviceHandler` interface aggregates the advice for collections of similar bytecodes into a single method. For example, one can advice just the `ICONST_0` and `ICONST_2` bytecodes, but the advice for both will be directed to the `VMAdviceHandler.adviseBeforeConstLoad` method. The aggregating methods do not provide a way to distinguish which bytecode generated the advice.

Recording Time

Certain handlers can optionally gather timing information, notably `SyncStoreVMAdviceHandler` and `VMLogStoreVMAdviceHandler`. The `VMATime` option provides a standard way to specify how time is gathered. If the option value is none, time is not recorded. If the value starts with `wallns`,

then wall clock time is gathered using `System.nanoTime`. If the value starts with `wallms`, then wall clock time is gathered using `System.currentTimeMillis`. `VMLogStoreVMAdviceHandler` requires the advice records to be ordered in the per-thread persistent stores, so that the analysis tool can reproduce a global order. As well as wall time, this can be achieved by a globally unique id, which can be enabled by setting the option value to `ida`. This variant has considerably less overhead that gathering wall clock time, yet tracks wall clock time quite closely. The `TimeTestVMAdviceHandler` can be used to compare the two values over a run.

Thread Advising

By default all application threads are enabled for advice. However a subset can be enabled by using the `VMATI` and `VMATX` options.

Sampled Advising

By default advising is on all the time which, evidently, has a significant performance impact on the application. Sampled advising, which is enabled by the -XX:VMASample option only enables advising periodically during the VM execution. The option value is a string of the form initialperiod,interval,period, all of which are optional. The interval value denotes the time, in milliseconds, between advising periods. The period value denotes the length of the advising period. The VM starts with advising enabled for initialperiod. If not specified the values default to 50, 50 and 10, respectively.

VM Options Summary

- `VMA`: boolean valued option that enables/disables advising in the VM. Default is `true`.
- `VMAMI=p`: `p` is a regular expression pattern specifying methods to include for instrumentation.
- `VMAMX=p`: regular expression pattern specifying methods to exclude for instrumentation. Overrides `VMAMI`.
- `VMAXJDK`: boolean valued option that excludes all JDK classes from instrumentation.
- `VMABI=p`: `p` is a regular expression pattern of bytecodes to include for instrumentation.
- `VMABX=p`: `p` is a regular expression pattern of bytecodes to exclude for instrumentation. Overrides `VMABI`.
- `VMAConfig=c`: `c` is a comma spearated list of configuration names that instrument for specific analyses.
- `VMATI=p`: `p` is a regular expression pattern of threads to have advising enabled.
- `VMATX=p`: `p` is a regular expression pattern of threads to have advising disabled. Overrides VMATI.
- `VMATime=none|wallns|wlaams|ida}`: specify how time is recorded in certain advice handlers. Default is `wallns`.
- `VMASample=initialperiod,interval,period`: run in sampling mode. Defaults to `50,50,10`.

7.21.4 Standard Handlers

Several existing handlers are provided, most notably handlers that store the generated advice to an external file for offline analysis. The default location for the external store is a directory vmastore in the current working directory that the VM is launched in. This can be changed by setting the `-Dmax.vma.store.dir` property.

`SyncStoreVMAdviceHandler` is a simple, but inefficient, handler that synchronously stores the advice record to the store, and incurs per-advice synchronization overhead as all threads access the same store. Output goes to the shared store file named vm in the store directory.

`VMLogStoreVMAdviceHandler` uses a custom instance of the `VMLog` class to store advice records in per-thread buffers. When the buffer is full it is flushed to a file in a compact textual format, that can be processed with the `QueryAnalysis` tool. `VMLogStoreVMAdviceHandler` adds a small, but generally consistent, overhead to the execution of each bytecode. Assuming the use of the default text-based store, it also causes object allocation as internal objects are converted into `String` representations when the records are flushed. The maximum latency occurs when the store buffer is flushed to the file. **Note:** this handler requires that some of its code to be included in the boot image as it adds some VM thread local variables, and these cannot be added at runtime. This handler operates in per-thread mode throughout the store process, thereby avoiding almost all synchronization overhead.

Developers can define additional handlers for specific purposes that may, for example, do all analysis internally in the VM. An example is `CBCVMAdviceHandler` that simply counts the advice calls and outputs a summary at the end of execution. `ThreadLocalVMAdviceHandler` analyses object use for thread locality. Evidently such handlers incur much less CPU overhead that those that externalize the data, but may incur additional memory costs. Handlers that only handle a subset of the advice calls should subclass `NullVMAdviceHandler` which defines a null implementation of each advice method. Note that `CBCVMAdviceHandler` can be used to estimate the size of the persistent store that would be created by `VMLogStoreVMAdviceHandler` or `SyncStoreVMAdviceHandler`, since the store will contain approximately the same number of lines as the number of advice counts reported. On average a trace line in the store is 16 bytes in length.

Per-Object State

One of the issues for analysis tools, either online or offline, is associating analysis-specific state with an object, for example a unique identifier. The standard approach is to use a `Map` but this has problems both with the space overhead and the fact that the map keeps objects reachable, which perturbs the behavior of the garbage collector. An WeakHashMap can mitigate the latter problem but perturbs the garbage collector in a different way and provides no guarantee of object lookup if the object is collected.

VMA provides a solution to this by using a custom Maxine object layout scheme, `XOhm`, to provide extra header words for use by VMA handlers. Evidently this has some impact on the behavior of the system, for example, making garbage collection more frequent owing to the increased object size. However, the overhead is as minimal as can be achieved. By default, one extra word is provided and basic access to the state word is provided by the `ObjectState` class. Support for unique identifiers is provided through the `ObjectId` interface and support for marking bits through the `ObjectBitSet` interface. The class `SimpleObjectState` implements both of these interface. Additional words can be included by setting the `max.vm.layout.xohm.words` system property on the image build. For example setting `-Dmax.vm.layout.xohm.words=2` would provide a total of two additional header words, one for use by `ObjectId` and `ObjectBitSet` and one for use by the handler. Access to the additional words is through the `ObjectVars` interface. The class `VarsObjectState` extends `SimpleObjectState` to implement this interface. Note that only scalar values can be stored in the extra header words as they are not scanned by the garbage collector.

Handlers that need to use an persistent object id to represent an object, should subclass `ObjectStateAdapter` which implements all the `VMAdviceHandler` methods that take `Object` types as arguments. Unique identifiers are assigned to objects returned by the NEW family of bytecodes. Objects passed as arguments to the other methods are checked for a uuid having been assigned and, if not, the abstract method `unseenObject`, which must be implemented the handler, is called. The adapter also handles unique id generation for `ClassLoader` instances, since these may be user defined.

7.21.5 Specifying handlers

Handlers can either be built into the boot image or loaded dynamically as a VM extension.

Building handlers into the boot image

To build an advice handler into the boot image set the `max.vma.handler.class system` property to the fully qualified name of a class that extends `VMAdviceHandler`, e.g.:

```
mx --Jp @-Dmax.vma.handler.class=com.oracle.max.vm.ext.vma.handler.store.
↪sync.h.SyncStoreVMAdviceHandler image @vma-t1x
```

A short form is available for the standard handlers using the max.vma.handler property:

- `null`: `NullVMAdviceHandler`
- `syncstore`: `SyncStoreVMAdviceHandler`
- `vmlogstore`: `VMLogStoreVMAdviceHandler`
- `cbc`: `CBCVMAdviceHandler`
- `tl`: `ThreadLocalVMAdviceHandler`

When creating a new handler it is important to prevent it being included in the boot image by default. This is achieved by adding a `Package` class to the handler package that specifies inclusion only when the `-Dmax.vma.handler.class` option matches the handler in question. See the the existing handlers for an example.

Dynamically loaded handlers

Follow the instructions for building a VM extension JAR file, using one of the included handlers as an example, the load the handler at runtime, e.g.:

```
mx vm -vmextension:yourhandler.jar ...
```

Also, look at one of the Eclipse JAR file creation descriptions for the existing handlers, in the file with extension jardesc. Note that all referenced classes that are not already included in the boot image must be specified in the jar file, as the VM extension mechanism has no search path support.

7.21.6 Instrumenting JDK classes in the Boot Image

It is possible to instrument JDK methods that were included in the boot image. This occurs transparently if such a JDK method is in set of methods specified to be instrumented. This is implemented on VM

startup by deoptimizing the methods in the boot image. Note that this can greatly increase the quantity of generated advice and also has an impact on the performance of the handlers themselves since, although advising is disabled in the scope of a handler, the JDK method is no longer optimized.

Note that the `-XX:+VMAXJDK` can be used to suppress instrumentation of all JDK methods. Setting this option is the easiest way to just instrument all application methods.

7.21.7 VMA and Handler Initialization

VMA overrides the standard run scheme, `JavaRunScheme`, with `VMAJavaRunScheme`, to interpose on the VM startup to perform VMA specific initialization. `VMAdviceHandler` defines an `initialise(MaxineVM.Phase phase)` method that normally should be overridden in a handler, and this method is called from `VMAJavaRunScheme`. The only phases that are of interest are `BOOTSTRAPPING`, `RUNNING` and `TERMINATING`. `BOOTSTRAPPING` is only relevant for the case where the handler is being included in the boot image, and provides an opportunity to allocate and initialize certain data structures in the boot heap. Other initialization must be performed in the `RUNNING` phase. In order for the handler to influence the behavior of the system, for example,to customize the bytecodes that are advised, the handler's initialise method is called before any methods are instrumented and before the JDK classes are considered for deoptimization and instrumentation. Note that this means that some JDK classes may be loaded and compiled without instrumentation as a side effect of being used by the handler's initialise method. However, these will be recompiled and instrumented if required as part of the JDK instrumentation phase. Actual advice method invocations are not enabled until after the `VMAJavaRunScheme.initialize` returns in the `RUNNING` phase. The `TERMINATING` phase is typically where the handler reports any results, using whatever mechanism it chooses.

Note that the `onLoad` method of a dynamically loaded handler is called in the `STARTING` phase, which precedes the `RUNNING` phase. Typically all the method should do is register an instance of the handler using `VMAJavaRunScheme.registerAdviceHandler`, so that it can be invoked in the `RUNNING` phase.

Properties to control VMA

VMA behavior can be controlled by setting some system properties in addition to the command line options. These are generally handler specific.

Boot Image Generation Properties

- `max.vma.vmlog`: This property must be set on a boot image build and controls whether the custom `VMLog` used by `VMLogStoreVMAdviceHandler` is included. It is on by default but the code can be excluded by setting the value to false.
- `max.vma.handler.class`: The fully qualified name of a handler class to be included in the boot image.
- `max.vma.handler`: The short name of a standard handler class to be included in the boot image.

Handler Specific Properties

- `max.vma.handler.cbc.sort`: When set, `CBCVMAdviceHandler` sorts the pan-thread advice counts by frequency.
- `max.vma.store.bufsize`: The size of the `StringBuilder` used to buffer store records. Default is 1 MB.
- `max.vma.store.flush`: The size at which the store buffer is flushed to the file; defaults to `max.vma.store.bufsize`.
- `max.vma.store.textkey`: Use a 3 character mnemonic key for stored records instead of a single digit code.

Performance

The performance overhead varies, evidently, with the set of bytecodes that are being advised and the set of classes that are subject to instrumentation. Performance is fundamentally limited by VMA currently being restricted to the T1X baseline compiler. The numbers presented below are for the SpecJVM98 benchmarks. The overhead of using T1X with no optimization for hot methods varies considerable depending on the application behavior, averaging 6.9% for SpecJVM98 with a range of 1.0% to 16.7%. The benchmarks that make use of the JDK classes show much less overhead since they are able exploit the optimized JDK methods compiled into the boot image.

The `NullVMAdviceHandler` provides a measure of the basic overhead of a handler. The overhead is relative to Maxine using T1X with no optimization of hot methods with every bytecode advised. The average overhead is 3.5%, with a range of 1.8% to 7.3%, when only the benchmark classes are instrumented. If we restrict the advice to just those bytecodes needed by the `objectuse` configuration, the overhead averages 2.1% with a range of 1.3% to 3.7%. If the JDK classes are also instrumented, including those in the boot image, the average overhead with all bytecodes advised increases to 5.6%, with a range of 1.8% to 12% and for the `objectuse` configuration averages 2.9% with a range of 1.4% to 6.8%.

7.21.8 Analysis Tools

QueryAnalysis

QueryAnalysis is a command line tool that was originally implemented in a project that was focused on analyzing objects for immutability. It reads the compact text form of the store file and builds a data structure suitable for analysis. The tool has no pre-defined analyses built in but supports dynamically loaded queries that are written to a standard API.

The basic data structure created by the tool is a list of the advice records in the file. Object instances and Maxine meta-objects, e.g. `MethodActor`, are replaced with types defined in the analysis tool, namely `ObjectRecord`, `ClassRecord`, `FieldRecord` and `MethodRecord`, with the obvious mappings.

The tools also builds some auxiliary data structures to facilitate analysis.

- `objects`: a map containing all the object instances in the store trace. The key is the id of the object and the value is the `ObjectRecord`. Note that since id's might be reused by the VM as objects are garbage collected, the id is tagged with the allocation epoch to ensure uniqueness.

- `classLoaders`: a map from classloader instances to a (sorted) map from classes loaded by that classloader. The key of the classLoaders map is the id of the classloader instance. The key of the class map is the name of the class and the value is the `ClassRecord`.
- `missingConstructors`: Inevitably object instances occur in the trace for which no allocation event was seen, e.g. objects allocated in the boot image. These are given id's that decrease from -1. The `missingConstructors` map is keyed by the id and the value is the associated `ObjectRecord`.
- `allocationEpochs`: a list of `AllocationEpoch` objects that define when garbage collection events have occurred. Each instance specifies the period from one garbage collection to the next.
- `objectCount`: the total number of object instances, that are not arrays, encountered in the trace.
- `arrayCount`: the total number of array instances in the trace.

Usage:

```
[-f inpath] [-v] [-i queryfile] [-e query]
```

The default value of `inpath` is `vmastore`. If the value is a directory (normal) it is expected to contain either a vm file or a set of per-thread files. If the value is a file, it is used directly.

Options:

- `-v`: Report on progress reading the store file, showing the time to read every 100000 records.
- `-e query`: initial query to execute
- `-i queryfile`: Execute the commands in queryfile before entering interactive mode.

After processing the store file the tool enters interactive mode allowing queries to be executed against the data structures described above. The input prompt is `%%`. The following interactive commands are available:

- `e query`: The tool prepends `com.oracle.max.vma.tools.qa.queries` and appends `Query` to queryclass and then attempts to load that class, which must be a subclass of `QueryBase`, and then invokes its execute method. E.g., e Foo will attempt to invoke `com.oracle.max.vma.tools.qa.queries.FooQuery.execute`. See below for details on the pre-defined queries.
- `i infile`: Execute commands from infile.
- `o outfile`: Redirect output to outfile or the standard output if outfile omitted.

A query is executed with the command:

```
%% e query
```

Standard Query Arguments

- `-v`: set verbose output mode.
- `-class name | -c name`: restrict output to class name
- `-thread name | -th name`: restrict output to thread name.
- `-id id`: restrict output to object with id.

- `-clid id`: restrict output to classloader id.
- `-abs`: report time as absolute

Evidently these arguments have a query-specific interpretation, within the general definition given above.

Pre-defined Analysis Queries

The analysis tool is meant to be easily extensible, but a set of simple query classes are included with the tool. Many of these derive from the prior work on immutability so, for example, when displaying objects it is typical for information on immutability to be output.

AdviceRecords -from fromindex -to toindex -showindex -indent

Lists the advice records reconstructed from the store. **Note:** Unless the range is constrained this generates even more output that contained in the original (comopressed) store, and is best redirected to a file with the o outfile command. The `-showIndex` option adds the index of the record as a prefix, and the `-indent` option indents on a method entry record.

BasicCounts

Displays the number of classes, classloaders, objects, arrays, the number of missing constructors and also displays the result of the `ImmutableCount` query.

Classes

Displays the classes, showing the classloader and the number of instances of each class.

Additional arguments:

- `-sortbycount`: sort the output from largest number of instances to smallest.

ClassLoaders

Displays the classloader objects in the trace, i.e., those that inherit from `java.lang.ClassLoader`.

DataByClassLoader

For each classloader, display the data on objects of a given class loaded by that classloader.

Example output:

```
Objects organized by classloader


Classloader: (sun.misc.Launcher$AppClassLoader) -1:0
  Objects organized by class
  test.Simple, total objects 1
```

(continues on next page)

(continued from previous page)

```
    1:0, c 223.318617ms, la 253.762114ms, lm 238.633649ms, stable for 49.
↪693585%
Classloader: (com.sun.max.vm.type.BootClassLoader) -2:0
  Objects organized by class
  java.io.PrintStream, total objects 1
    -4:0, c 253.77319ms, la 253.777241ms, lm 253.77319ms, stable for 100%
etc.
```

The end of construction by c, the last access time by la, and the last modify time by lm. Note that in the current implementation objects whose construction was not observed are given ids that decrease from -1, and a creation time of the first time they are accessed.

Additional options:

- `-showthread`: show the thread that allocated the instance (default `false`)
- `-sort_lt`: Sort instances by lifetime (highest to lowest)
- `-sort_mlt`: Sort instances by mutable lifetime (highest to lowest)
- `-summary`: Suppress the individual instance output and replace with the percentage of objects immutable for more than a given percentile (default `100`) (default `false`).
- `-pci percentile`: Set the percentile to use with -summary.
- `-sort_summary mode`: The summary data can be sorted by class `mode == class`, total number of instances `mode == total` or total number immutable for more than percentile, `mode = imm_total`.

DataByClass

The same output as `DataByClassLoader` except for all classes, irrespective of classloader.

DataByObject

The data, in same format as `DataByClass` on all the objects in the objects map.

DataOnObject

The data on a specific object id.

DataByThreads

Similar output to `DataByClass` except grouped by the allocating thread.

Additional arguments:

- `-summary`: restrict output to the total number allocated and the total live number.
- `-sort_lt`: as per `DataByClass`.
- `-sort_mlt`: as per `DataByClass`.

GC

Displays the allocation epochs in the trace, that is, the periods between garbage collections. If `-r` is set, also displays the objects that were collected at the end of each epoch.

ImmutableClassBuckets

For each class: first computes the immutable lifetime as a percentage, then counts how many objects fall into buckets that are 1% in size. E.g.:

```
Class sun.reflect.NativeMethodAccessorImpl, Object count 4

  4: 1 (25.00)  42: 1 (25.00)  75: 1 (25.00)  100: 1 (25.00)
```

This means that one object was immutable for 4%, one for 42%, one for 75% and 1 for 100% of respective lifetime. The number in brackets is the percent of the total for that bucket.

ImmutableCount

Displays the immutable object percentage and the immutable array percentage.

LiveObjects

Displays the total number and size of the live objects.

MissingConstructor

Displays the number of objects for which no trace was generated for construction and, if `-v` is set, the data on those objects.

MutableObjects

Display all the mutable objects of a given class and the list of modifications.

StaticFieldAccess

Displays the accesses to the static fields of classes.

ThreadLocal

Analyzes the data for thread locality, reporting on objects created by one thread that are accessed by another thread. Evidently, this analysis is dependent on the appropriate bytecodes being traced. The `objectuse` configuration will report any use of an object, whereas `objectaccess` will report an actual access to the content of an object.

CallGraph

Assuming that the store file contains method entry and exit traces, this query constructs a call graph and displays it graphically using a standard Swing JTree.

ConvertLog

This tool can perform various conversion operations on the store file.

Usage:

```
[-f inpath] [-o outfile] [-readable | -unbatch | -batch | -merge] [-
↪abstime]
```

The default value if `inpath` is `vmastore`. If no output file is specified the output goes to the standard output. The options have the following effect:

- `-readable`: Generate a (more) readable form of the store file.
- `-unbatch`: Convert the store file containing records batched by thread into a time ordered file
- `-batch`: Reverse the process performed by `-unbatch`. This is a debugging tool to check the `-unbatch` command.
- `-abstime`: By default store files denote time by the increment between successive records (in a batch). This option will convert the file to use absolute time for each record.
- `-merge`: Merges all per-thread store files into a single output file.

Simple Tests

The `com.oracle.max.vm.ext.vma.test` package contains some simple test programs to exercise the system, of which we highlight a few here. To simplify the exposition, assume that the current working directory contains the Maxine projects and the following aliases have been defined:

```
alias qa="java -classpath com.oracle.max.vma.tools/bin:com.oracle.max.vm.
↪ext.vma/bin com.oracle.max.vma.tools.qa.QueryAnalysis"
alias maxvma="max vm -cp com.oracle.max.vm.ext.vma/bin"
```

Simple

This is an example of a class with a non-final field that is *stable* in the sense that it is written to once shortly after the constructor executes and is then immutable from that point on. To demonstrate this, execute:

```
$ maxvma '-XX:VMAMI=test.Simple.*' test.Simple
$ qa
%% e DataByClass -c test.Simple
Objects organized by class
test.Simple, total objects 1, cl: (sun.misc.Launcher$AppClassLoader) -1:0
  1:0, c 223.318617ms, la 253.762114ms, lm 238.633649ms, stable for 49.
↪693585%
```

GCTest

GCTest is a multi-threaded program where each thread iteratively builds up a list of objects with a randomly generated lifetime, removing those that have expired after each iteration, and then invoking `System.GC`. E.g.,

```
$ maxvma -XX:VMAMI='test.GCTest.*' test.GCTest
$ qa -v

processing trace file vmastore starting
processed 100000 traces in 1331 ms (1331)
processed 200000 traces in 1887 ms (556)
processed 300000 traces in 2399 ms (512)
processed 400000 traces in 2702 ms (303)
processed 500000 traces in 3024 ms (322)
processing trace file vmastore complete
%% e GC
Allocation epochs
Epoch 0, 0ms, 646.481ms
Epoch 1, 646.481ms, 772.329ms
Epoch 2, 772.329ms, 879.328ms
Epoch 3, 879.328ms, 985.426ms
Epoch 4, 985.426ms, 1117.057ms
Epoch 5, 1117.057ms, 1226.683ms
etc.
```

Since the store file is considerably larger for this example, we used the `-v` option to report processing speed.

ThreadLocal01

ThreadLocal01 is a multi-threaded program where several worker threads (default 2) build lists of objects that are private to the thread. Optionally the program can be configured so that they leak objects that are accessed by a leak observer thread. To run in private (thread local) mode, execute:

```
$ maxvma -XX:VMAMI='test.ThreadLocal.*' test.ThreadLocal01

running with 2 threads
Thread Generator-0 running
Thread Generator-1 running
Thread Generator-0 returning
Thread Generator-1 returning
global list size 22, LeakObserver accessCount 437
main thread terminating
$ qa
%% e ThreadLocal
Check objects allocated by thread Generator-1
Check objects allocated by thread Generator-0
Check objects allocated by thread main
object 5:0 is accessed by thread Generator-0
object 2:0 is accessed by thread Generator-1
object 2:0 is accessed by thread Generator-0
object 9:0 is accessed by thread Generator-1
object 8:0 is accessed by thread Generator-1
object 6:0 is accessed by thread Generator-0
```

The ThreadLocal query determines that objects allocated by the generator threads are not accessed by any other thread.

Now execute:

```
$ maxvma -XX:VMAMI='test.ThreadLocal.*' test.ThreadLocal01 -l

running with 2 threads
Thread LeakObserver running
Thread Generator-0 running
Thread Generator-1 running
Thread Generator-0 returning
Thread Generator-1 returning
main thread terminating
$ qa
%% e ThreadLocal
Check objects allocated by thread Generator-1
object 120:0 is accessed by thread LeakObserver
object 120:0 is accessed by thread Generator-0
object 161:0 is accessed by thread LeakObserver
object 283:0 is accessed by thread Generator-0
object 387:0 is accessed by thread LeakObserver
object 381:0 is accessed by thread LeakObserver
object 137:0 is accessed by thread Generator-0
object 147:0 is accessed by thread LeakObserver
object 30:0 is accessed by thread LeakObserver
object 30:0 is accessed by thread Generator-0
object 62:0 is accessed by thread LeakObserver
object 51:0 is accessed by thread LeakObserver
object 406:0 is accessed by thread LeakObserver
object 347:0 is accessed by thread LeakObserver
object 309:0 is accessed by thread Generator-0
object 208:0 is accessed by thread LeakObserver
object 208:0 is accessed by thread Generator-0
object 202:0 is accessed by thread LeakObserver
etc.
```

In this case the query detects that objects created by `Generator-1` have leaked to both `LeakObserver` and `Generator-0`. **Note:** since both generator threads are leaking objects to the same global list, they also see each others leaks.

Note that a similar analysis is performed online by the ThreadLocalVMAdviceHandler.

Bugs and Limitations

The main limitation at present is that VMA is only available for the baseline (JIT) compiler, and that many bytecodes do not support both BEFORE and AFTER advice.

7.22 VM Operations

A VM operation, implemented by class `com.sun.max.vm.runtime.VmOperation` is an operation that can be performed on one or more target threads by the `VmOperationThread` VM operation thread. In the normal case a VM operation is performed after the target threads are frozen at a *safepoint*, in which case every frame of a compiled/interpreted method on a frozen thread's stack is guaranteed to

be at an execution point where the complete frame state of the method is available. A VM operation is formed by creating an instance of a subclass of `com.sun.max.vm.runtime.VmOperation` and invoking the submit method on the instance. The behavior of the operation is specified by overriding the `doThread` method which, by default does nothing. I.e, such a default operation would simply freeze the threads, do nothing, and then release them. The details of the relationship between the `VmOperation` thread and the target threads is specified by an instance of the `VmOperation.Mode` class. The normal case is indicated by `Mode.Safepoint`, which causes all target threads to be frozen and the `VmOoperation` thread to block until the operation completes. We will focus on the normal case in what follows and defer discussion of the other, more unusual modes, until later. Note that the `VmOperation` class is only intended for use within the VM implementation. Consequently, in the interface, threads are specified by the `com.sun.max.vm.thread.VmThread` class and not `java.lang.Thread`.

A VM operation may target a subset of the threads in the VM, the degenerate case being a single thread, which is specified explicitly in the constructor for `VmOperation` by the `singleThread` argument. If this value is not null it denotes an operation solely on the specified thread. If `singleThread` is null it specifies a multi-thread operation. This design simplifies the single thread case and avoids having to provide the set of target threads explicitly in the multi-thread case. By default the multi-thread variant acts on all threads (except the `VmOperation` thread itself). However, the `VmOperation` instance can provide finer control by overriding the `operateOnThread` method. If this method returns false for any `VmThread` passed as argument, that thread is ignored. I.e. it will neither be frozen nor have the operation performed.

The following is a trivial example that simply prints the name of each frozen thread.

```
VmOperation op = new VmOperation("Example", null, Mode.Safepoint) {
    @Override
    public void doThread(VmThread vmThread, Pointer ip, Pointer sp,␣
↪Pointer fp) {
        System.out.println("Thread " + vmThread.getName() + " is stopped␣
↪at " + Long.toHexString(ip.toLong()));
    }
op.submit();
```

The first argument to the constructor is only used when tracing the operation for debugging purposes, which can be enabled with the VM command line argument `-XX:+TraceVmOperations`. The three `Pointer` arguments to `doThread` are the code address at which the thread is stopped and the stack pointer and frame pointer, respectively. These values are generally used in operations that need to access the execution stack, for example to generate a stack trace. Note that these values are all guaranteed to be valid, in particular a thread that has been started but is not yet executing Java code will be filtered out and not passed to `doThread`. Note that the constructor only initializes the instance, no part of the operation occurs until the submit method is invoked. **TBA:** can a VMOperation instance be reused?

Many internal operations within the VM are implemented using `VMOperation`, most notably garbage collection. So what happens if the code in `doThread` allocates memory and happens to cause a garbage collection? Note that it can be quite difficult to determine by inspection whether a method allocates objects. The above example contains no new keywords, but the string concatenation does so implicitly. In fact it is very difficult to write allocation free code, so for `VmOperation` to be useful there must be a solution to what is in effect a nested `VmOperation`. Since most operations can cause a garbage collection, `VmOperation` supports nested operations by default, but they can be disabled by invoking the more general constructor that takes a `disAllowNestedOperations` argument.

The output from running the above example should look like this:

```
Thread Signal Dispatcher is stopped at 0x103fcb029
Thread Finalizer is stopped at 0x103fb2c44
Thread Reference Handler is stopped at 0x103fb2c44
Thread main is stopped at 0x103fb2c44
```

Notice that three non-application (system) threads are included in the list. Note also that all but `Signal Dispatcher` are stopped at the same address. It doesn't matter how many time you run the application, this will always be the case. The reason related to the mechanism that is used to freeze the threads, and is explained in the section on implementation details.

What if we only wanted the `VmOperation` to operate on application threads? One way, although not a very stable solution, would be to provide an `Override` for `operateOnThread` that compared the textual names of the threads. A better way would be to exploit the fact that system threads exist in a separate `ThreadGroup` from application threads. For example, this `operateOnThread` method would do:

```
protected boolean operateOnThread(VmThread vmThread) {
    if (!systemThreads) {
        return vmThread.javaThread().getThreadGroup() != VmThread.
↪systemThreadGroup;
    } else {
        return true;
    }
}
```

7.22.1 Implementation Details

A thread is frozen at a safepoint when it is blocked in native code (typically on an OS-level lock) and cannot (re)enter compiled/interpreted Java code without being thawed (see class `ThawThread`) by the VM operation thread.

Freezing a thread is a co-operative action between the VM operation thread and the thread(s) being frozen. There are two alternative implementations of this mechanism provided. The first uses atomic instructions and the second uses memory fences. They are named `CAS` and `FENCE` and are described further below.

CAS

Atomic compare-and-swap (CAS) instructions are used to enforce transitions through the following state machine:

```
+------+                              +--------+                                ␣
↪    +----------+
|      |--- M:JNI-Prolog{STORE} --->|        |--- VM:WaitUntilFrozen{CAS} -
↪-->|          |
| JAVA |                              | NATIVE |                                ␣
↪    | FROZEN   |
|      |<--- M:JNI-Epilog{CAS} -----|        |<----- VM:ThawThread{STORE} -
↪---|          |
+------+                              +--------+                                ␣
↪    +----------+
```

The syntax for each transition operation is:

```
thread ':' code '{' update-instruction '}'
```

The state pertains to the mutator thread and is recorded in the thread local variable of the mutator thread. Each transition describes which thread makes the transition (`M` is the mutator thread, and `VM` is the VM operation thread), the VM code implementing the transition `JNI-Prolog`, `JNI-Epilog`, `WaitUntilFrozen` and `ThawThread` and the instruction used to update the state variable (`CAS` is atomic compare-and-swap and `STORE` is normal memory store)

FENCE

Memory fences are used to implement Dekkers algorithm to ensure that a thread is never mutating during a GC. This mechanism uses both the `MUTATOR_STATE` and `FROZEN` thread local variables of the mutator thread. The operations that access these variables are in `Snippets.nativeCallPrologue()`, `Snippets.nativeCallEpilogue()`, `WaitUntilFrozen` and `ThawThread`.

The choice of which synchronization mechanism to use is specified by the `UseCASBasedThreadFreezing` variable.

Freezing a thread requires making it enter native code. For threads already in native code, this is trivial, i.e., there's nothing to do except to transition them to the frozen state. For threads executing in Java code, *safepoints* are employed. Safepoints are small polling code sequences injected by the compiler at prudently chosen execution points. The effect of executing a triggered safepoint is for the thread to trap. The trap handler will then call a specified `AtSafepoint` procedure. This procedure synchronizes on the global GC and thread lock. Since the VM operation thread holds this lock, a trapped thread will eventually enter native code to block on the native monitor associated with the lock.

This mechanism is similar to but not exactly the same as the `@code VM_Operation` facility in HotSpot except that Maxine `VmOperations` can freeze a partial set of the running threads as Maxine implements per-thread safepoints (HotSpot doesn't).

Implementation note

It is simplest for a mutator thread to be blocked this way. Only under this condition can the GC find every reference on a slave thread's stack. If the mutator thread blocked in a spin loop instead, finding the references in the frame of the spinning method is hard (what refmap would be used?). Even if the VM operation is not a GC, it may want to walk the stack of the mutator thread. Doing so requires the VM operation thread to be able to find the starting point for the stack walk and this can only reliably be done (through use of the Java frame anchors) when the mutator thread is blocked in native code.

7.22.2 Suspend and Resume Thread Operations

The ability to suspend and resume threads, which is required by the JVMTI interface, is implemented using `VmOperation`, and nested classes `SuspendThreadSet` and `ResumeThreadSet` are provided in `VmOperation`. These operations are also used by the (deprecated) methods `Thread.suspend` and `Thread.resume`.

A normal VM operation suspends (freezes in `VMOperation` terminology) the thread set temporarily, runs the operation, and then resumes the thread set. All the machinery to safepoint a running Java thread or handle a thread in native code is appropriate for the suspend operation, but the thread must stay suspended after the operation completes until the resume operation is invoked. Ordinarily a thread is frozen either by blocking on the `THREAD_LOCK` monitor held by the `VmOperationThread` (thread

in Java) or spinning in the return sequence from native code (thread in native). Evidently the monitor must be released to exit the `VmOperation` so an additional mechanism is necessary to actually suspend (as opposed to freeze) the thread. Consider the two cases:

1. Thread in Java: The thread is blocked on the `THREAD_LOCK` monitor, called from the trap handler that handled the safepoint. Note that because it is blocked on the monitor, it is also actually in native code. The entire monitor acquisition process, which in Maxine currently can comprise several stack frames, must be unwound in order to release the `THREAD_LOCK` monitor. In fact we unwind all the way back to the trap handler.
2. Thread in Native: There are actually two cases here. Either the thread is truly blocked in native code, for example, on some other monitor or performing I/O, or it is caught in the native code return sequence and is spinning waiting to be unfrozen. In either case, when the thread actually returns it must then suspend (unless a resume occurs before the thread actually returns).

A thread is marked for suspend by setting bit zero in the `SUSPEND` field of the `VmThreadLocal` area. This value is only ever written while the thread is frozen in the body of the `VmOperation.SuspendThreadSet` or `VmOperation.ResumeThreadSet` operation. When a thread is unfrozen it will promptly check the `SUSPEND` bit and if it is set, will actually suspend on a native OS monitor (suspend monitor) that is pre-allocated to every thread. For a thread in native this check happens as the final act of the native return epilogue. To handle the special case of a thread that was safepointed and is executing that sequence to release the `THREAD_LOCK` monitor, bit 1 of the `SUSPEND` field is also set for safepointed threads, and the native epilogue checks that bit and does not suspend.

The Resume operation clears the `SUSPEND` field in the `VmOperation` body and notifies the suspend monitor, which will cause any thread that actually suspended to become runnable again. Note that a resumed thread must recheck the `SUSPEND` field since it is possible that another suspend operation occurred before the thread actually got on CPU.

7.23 VM Tooling interface

As a research VM, Maxine should make it is easy as possible to analyze the behavior of applications and Maxine itself. Since Maxine is meta-circular, many of the techniques used to analyze applications are also applicable to Maxine, although some aspects of meta-circularity can cause problems that can be hard to foresee, such as trying to allocate on the heap during a garbage collection.

Maxine supports (most of) the standard JVMTI API, which supports agents written in native code. In particular, the standard jdwp agent is supported which allows debugging of applications and, experimentally, the VM itself, with a Java IDE. Since Maxine is written in Java, the native JVMTI interface is not very appropriate, and Maxine provides a Java version of the JVMTI API, that is much easier to use than the native interface.

7.23.1 VMTI

Maxine does not make reference to any specific tooling interface or implementation in the core VM code. Instead it defines an interface VMTI that defines methods that a tooling implementation must implement to be included in Maxine. Multiple tooling implementations can be active in a single VM. Currently Maxine supports two tooling implementations, JVMTI and Virtual Machine Level Analysis. The latter, which overlaps somewhat with JVMTI, is specific to Maxine and primarily supports the advising of the execution of the virtual machine at the bytecode abstraction level.

7.23.2 JVMTI

The JVMTI implementation is contained in the package `com.oracle.max.vm.ext.jvmti`. Although separately specified in an extension package, currently it is included by default in the default boot image. The implementation is incomplete but sufficient for many purposes, including debugging. Notable omissions are the methods related to monitor contention and the reference walking heap iterators. The implementation will be completed in due course.

7.23.3 JJVMTI

JJVMTI is a Java version of the standard JVMTI native interface. As far as possible it is equivalent the native version, so translation between the two should be straightforward. Some design choices were changed to reflect the nature of Java. For example, whereas JVMTI returns errors as the function result, and uses pointers to caller defined variables to pass data, JJVMTI throws an exception in the event of an error and returns data as the method result. Also, whereas JVMTI necessarily uses either JNI handles or scalar values to represent classes and methods, JJVMTI uses Maxine's actor classes, e.g., `ClassActor`
Using JJVMTI

Writing a JJVMTI agent is considerably simpler than writing the equivalent JVMTI native agent, as there is no need to deal with all the complexity of the JVMTI and JNI native interfaces. However, since the agent must necessarily access Maxine VM classes, it must either be included in the boot image or dynamically loaded as a VM extension (preferred). Note that, unlike JVMTI native agents, JJVMTI agents cannot get control early in the VM startup, so certain changes to the VM environment cannot be made. This currently does not affect Maxine as it does not reconfigure itself in response to JVMTI capability requests.

The standard form of an agent that can be included in the boot image or loaded dynamically is as follows:

```
public class Agent extends NullJJVMTICallbacks {

    private static Agent agent;
    private static String AgentArgs;

    static {
        agent = (Agent) JJVMTIAgentAdapter.register(new Agent());
        if (MaxineVM.isHosted()) {
            VMOptions.addFieldOption("-XX:", "AgentArgs", "arguments for␣
↪exemplar JJVMTI agent");
        }
    }

    /***
     * VM extension entry point.
     * @param args
     */
    public static void onLoad(String agentArgs) {
        AgentArgs = agentArgs;
        agent.onBoot();
    }

    /**
     * Boot image entry point.
     */
    @Override
```

(continues on next page)

(continued from previous page)

```
    public void onBoot() {
        agent.setEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_VM_INIT,␣
↪null);
    }

    @Override
    public void vmInit() {
        if (AgentArgs != null) {
            // process arguments and enable needed JVMTI capabilities
        }
    }
}
```

Note that the mechanism for communicating arguments to the agent is necessarily different between a boot image agent and a dynamically loaded agent. In the former case a new VM command line option is defined, whereas in the latter case the arguments are passed in the VM extension option, `-vmextension:jar[=args]`. Note also that onBoot is only called in boot image mode and onLoad is only called when dynamically loaded, as per the specification for VM extensions. Since the VM is still in `PRIMORDIAL` mode in `onBoot` the recommended idiom is to enable the `VMINIT` event and do all further processing in the `vmInit` event callback, which is invoked (if enabled) in either case.

An example of a complete JJVMTI agent is the conversion of the native heap viewer agent that is supplied as a demo with the JDK. This agent also demonstrates one of the meta-circularity issues with JJVMTI. The `heapIteration` callback is not callback safe in JVMTI terminology, in particular, it cannot allocate. However, when the agent is dynamically loaded, Maxine will attempt to allocate implicitly as the `heapIteration` method will be compiled at the point that it is first invoked. This is finessed by the agent by forcing the compilation of `heapIteration` in the `vmInit` method. Note that this is not an issue if the agent is included in the boot image owing to ahead of time compilation. It would also be mitigated if Maxine kept a separate VM and application heap.

7.24 The Maxine Inspector

The Maxine Inspector is the essential all-in-one companion tool for Open Source Maxine VM development. It plays many roles:

- object, class, and method browser;
- code views include disassembled machine code, disassembled bytecode, and source;
- low-level debugger (imagine gdb or dbx) with visual displays of threads, registers, stacks, stack frames, thread local values, breakpoints, memory watchpoints, etc.;
- intermediate representation debugger;
- source-level Java debugger (eventually); and
- serviceability agent (eventually).

This short (5 minute) 2009 video describes the goals of the inspector and highlights some of the ways in which it makes Maxine VM development highly productive.

Please be aware that the Inspector is very much a work in progress, as is the Maxine VM itself. The two have co-evolved and will continue to do so as the design of the VM matures and the concerns of the developers expand. Functionality is constantly being improved and extended, so there are already places where the current system differs from what you will find here.

More discussion and more detailed documentation follows below.

7.24.1 Goals

In addition to enhancing the productivity of our own development team, the Maxine Inspector is part of our strategic goal of making VM experimentation more accessible to a wider audience. By leveraging the meta-circularity of the VM itself (and sharing a great deal of the VM's source code), the Inspector makes it possible to visualize concisely many aspects of VM state that are elusive and widely distributed in other systems. These same advantages also make it possible to debug the VM with a single tool, highly specialized for this purpose.

7.24.2 Background and Rationale

Debugging virtual machines (VMs) presents unique challenges, expecially for a meta-circular VM, such as Maxine, that is self-implemented in the same language it implements. Making sense of Maxine's runtime state requires interaction simultaneously at the source, bytecode, and machine code abstraction levels, and it must leverage knowledge of the VM's design. Specific issues include:

- Maxine VM code must be largely optimized statically, not only for ordinary performance reasons, but also in order to be able to bootstrap the VM at all.
- Dynamic optimization at runtime can be applied to the VM's own implementation, not just application code.
- Mapping optimized code locations back to bytecode and source locations is not generally possible without onerous limitations.
- Dynamically de-optimizing code for debugging can be effective for application code, but only when the VM can be assumed correct.
- Debugging the VM itself, however, requires scrutinizing its lowest-level, most optimized code representations and runtime machine state.
- Special considerations arise when debugging garbage collection, for example the Inspector's dependence on the VM's meta-information about classes, methods, etc., which are represented as ordinary *Java objects* in the heap; garbage collection, however, routinely breaks the heap invariants that make those objects accessible.
- Good debugging support is paramount for a VM intended for experimentation and fast prototyping.
- The Maxine Inspector addreses these concerns, supporting comprehensive inspection and interaction at all program representation levels.
- Finally, these services must be available to developers in a wide variety of contexts: examining a pre-execution *boot image*, examining and controlling a live VM process (local or remote), and post-mortem examination of a VM core dump.

Furthermore, the Inspector's design exploits the fact that it is implemented in the same language that the VM implements and is implemented in; this gives rise to many code reuse opportunities. For example, the same mechanisms used by the VM's *boot image generator*, which allow the creation of objects in the binary runtime format for a potentially different platform, can be used by the Inspector to examine binary runtime state for a potentially different platform than the Inspector's host.

7.24.3 Downloading and Building the Inspector

The Inspector source code is part of the Maxine VM repository. It will *download and build* automatically with the rest of the Maxine code. See also Inspector-specific issues on *various platforms*.

7.24.4 A Tour Through The Maxine Inspector

The best way to learn about the Inspector (and about many aspects of the Maxine VM) is to start up the Inspector on a simple VM session. For a beginner's introduction, however, the following pages introduce specific aspects of the Inspector's operation, in some cases with short video segments. The topics are threaded in a sequence so that you can navigate through them in order if you are new to the Inspector.

Boot Image Configuration

A good introduction to some of the Maxine VM's architectural features is provided by the Inspector command via the Boot Image info entry on any *View* menu. This produces an Inspector window displaying configuration parameters of the boot image being inspected. The boot image and its configuration can be inspected with or without a running VM process.

This short 2008 video demonstrates this view, although some evolution has taken place since then.

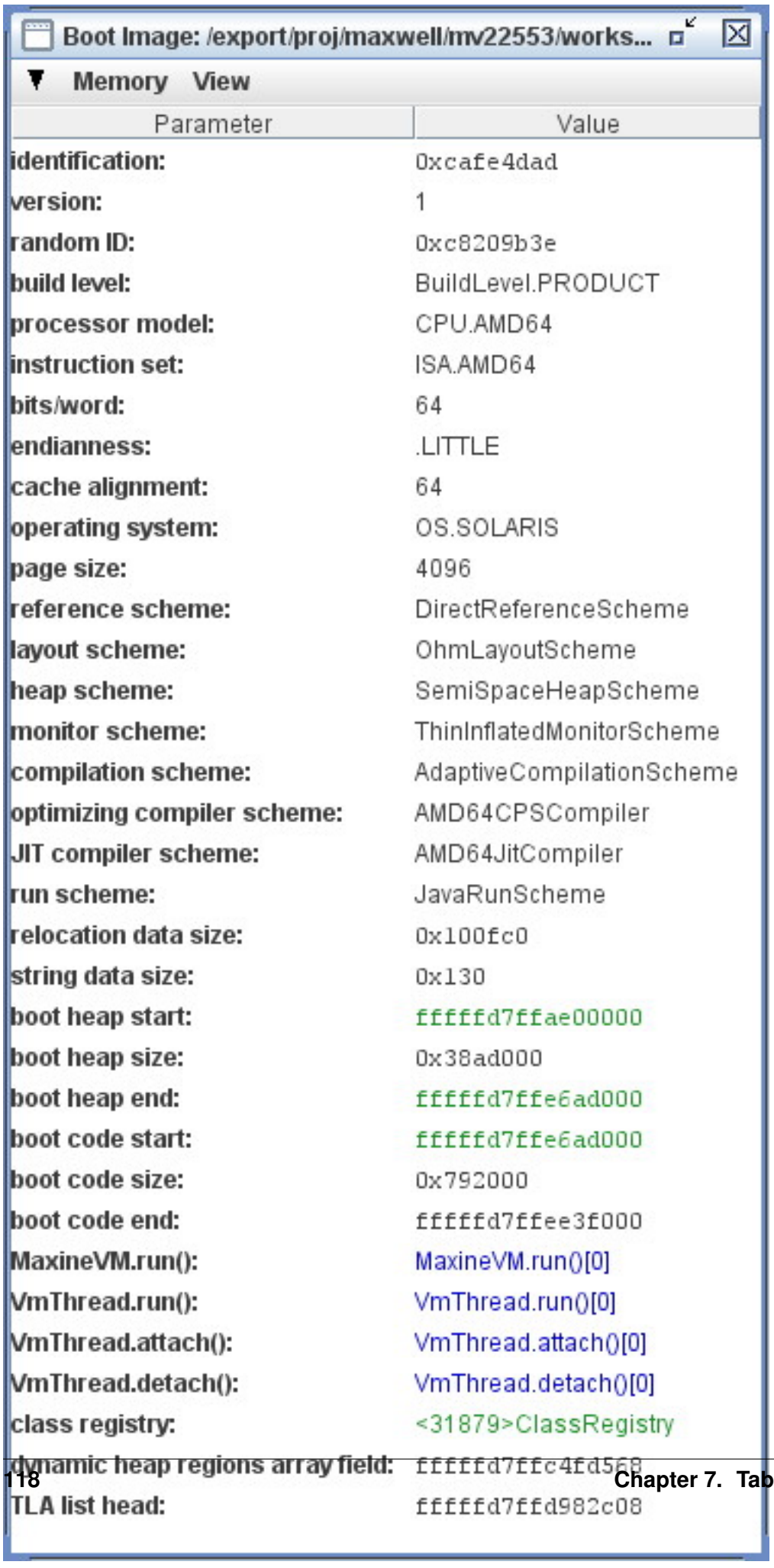
A more detailed description of the display appears below.

The Boot Image Inspector

The boot image contains several groups of configuration parameters, each of which relates to some aspect of the Maxine implementation. The Boot Image Inspector displays them in a simple tabular format, with entries in several general categories:

- *Identification* of the particular boot image build.
- Basic *build options*, e.g. `DEBUG` or `PRODUCT`.
- *Target machine properties*: the model, instruction set, word size, endianness, etc. for which code is compiled, both in the boot image and at run time.
- *Operating System properties*.
- *Maxine schemes*: pluggable modules that implement specific functions in the VM. For example, the *grip scheme* implements low level memory addressing, at which level garbage collection takes place; the *run scheme* directs what happens at VM startup, which could be running a standard Java program, as in the example, but could something else specified at build time.
- Parameters describing the *boot heap*: a pre-populated heap segment containing objects created at build time, in the same format as the dynamic heap segments created at run time.
- Parameters describing the *boot code* region of memory, which contains compiled code in the same format as the regions of compiled code that are created by dynamic compilation and recompilation at run time.
- *Code entry pointers*: specific addresses in the boot code region (displayed symbolically by the inspector in the example) for distinguished methods that will be called at VM startup.

- *Distinguished object pointers*: specific addresses in the boot heap region for objects of importance at VM startup, for example the root `ClassRegistry` object (displayed symbolically by the inspector in the example).


Boot Image: /export/proj/maxwell/mv22553/works...
Memory View
Parameter
Value
identification: 0xcafe4dad
version: 1
random ID: 0xc8209b3e
build level: BuildLevel.PRODUCT
processor model: CPU.AMD64
instruction set: ISA.AMD64
bits/word: 64
endianness: .LITTLE
cache alignment: 64
operating system: OS.SOLARIS
page size: 4096
reference scheme: DirectReferenceScheme
layout scheme: OhmLayoutScheme
heap scheme: SemiSpaceHeapScheme
monitor scheme: ThinInflatedMonitorScheme
compilation scheme: AdaptiveCompilationScheme
optimizing compiler scheme: AMD64CPSCompiler
JIT compiler scheme: AMD64JitCompiler
run scheme: JavaRunScheme
relocation data size: 0x100fc0
string data size: 0x130
boot heap start: ffffffd7ffae00000
boot heap size: 0x38ad000
boot heap end: ffffffd7ffe6ad000
boot code start: ffffffd7ffe6ad000
boot code size: 0x792000
boot code end: ffffffd7ffee3f000
MaxineVM.run(): MaxineVM.run()[0]
VmThread.run(): VmThread.run()[0]
VmThread.attach(): VmThread.attach()[0]
VmThread.detach(): VmThread.detach()[0]
class registry: <31879>ClassRegistry
dynamic heap regions array field: ffffffd7ffc4fd568
TLA list head: ffffffd7ffd982c08

As with many data displays in the Inspector, the items in the Value column have additional useful behavior. For example, most provide additional information about the displayed value in a mouse-over "Tooltip" display that appears when the mouse hovers over the display. In simple cases, such as integers, the Tooltip might display the value in another base. For example the page size item displays in decimal by default, but the hexadecimal value appears in the Tooltip. Conversely, the boot heap size displays in hexadecimal by default, and the decimal value appears in the Tooltip.

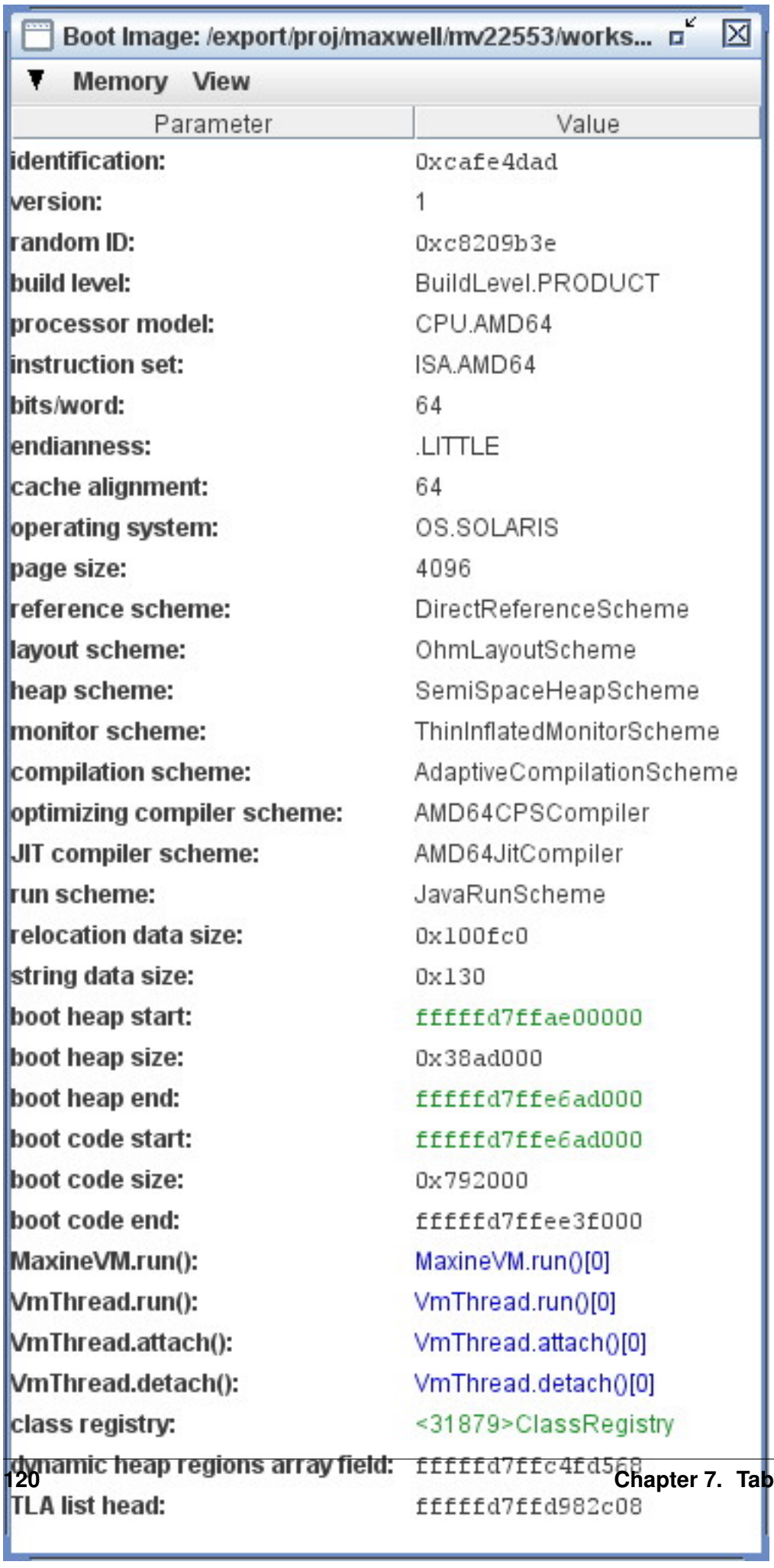
Any display item showing a memory value that could be interpreted as a pointer to a memory location exhibits much more complex behavior, described in more detail in *Memory Word Values*. The Inspector investigates each of these values empirically to determine where such a value might point in the VM's current memory. In the displayed example, the value of the parameter `boot heap start` was discovered to point at a heap object, presumably the first object in the region. Although displayed in hexadecimal by default, the item is color coded green to reveal this fact, and an alternate display showing information about the object (for example the `class registry` pointer) might also appear by default.

Similarly, parameter named `MaxineVM.run()` was discovered to point to the compiled code for a specific method, in this case evidently the correct one; in the example, these are displayed symbolically by default. These display items also exhibit dynamic behavior in response to various mouse actions. For more detail, see *Memory Word Values*.

An optional *Memory Regions Column* is available by selecting the *View Options* entry from the *View* menu. This setting is persistent, and it can also be set as a User Preference.

Memory Word Values

Many Inspector views display values that represent the contents of a memory word in the VM. Such words might contain primitive data values, but they also might contain addresses that point to other locations in the VM's memory such as heap objects and executable instructions. We call an Inspector element that displays the contents of a memory word a *Memory Word Value*. For example, in the Boot Image Inspector, shown here, of the parameter values in the lower part of the display are such Memory Word Values.


Boot Image: /export/proj/maxwell/mv22553/works...
Memory View
Parameter
Value
identification: 0xcafe4dad
version: 1
random ID: 0xc8209b3e
build level: BuildLevel.PRODUCT
processor model: CPU.AMD64
instruction set: ISA.AMD64
bits/word: 64
endianness: .LITTLE
cache alignment: 64
operating system: OS.SOLARIS
page size: 4096
reference scheme: DirectReferenceScheme
layout scheme: OhmLayoutScheme
heap scheme: SemiSpaceHeapScheme
monitor scheme: ThinInflatedMonitorScheme
compilation scheme: AdaptiveCompilationScheme
optimizing compiler scheme: AMD64CPSCompiler
JIT compiler scheme: AMD64JitCompiler
run scheme: JavaRunScheme
relocation data size: 0x100fc0
string data size: 0x130
boot heap start: ffffffd7ffae00000
boot heap size: 0x38ad000
boot heap end: ffffffd7ffe6ad000
boot code start: ffffffd7ffe6ad000
boot code size: 0x792000
boot code end: ffffffd7ffee3f000
MaxineVM.run(): MaxineVM.run()[0]
VmThread.run(): VmThread.run()[0]
VmThread.attach(): VmThread.attach()[0]
VmThread.detach(): VmThread.detach()[0]
class registry: <31879>ClassRegistry
dynamic heap regions array field: ffffffd7ffc4fd568
TLA list head: ffffffd7ffd982c08

Memory Word Values are among the most important aspects of the Inspector, and they appear in almost every kind of view. They exhibit a variety of useful behaviors, described on this page, designed to make the Inspector as useful as possible.

Investigating memory references

A Memory Word Value is often bound to a specific word location in the memory of a running VM. After each VM execution cycle, the Inspector "refreshes" every Memory Word Value, which causes the value in each word to be from memory read again. Each time this happens, the Inspector attempts to relate the value found to other information that is already known about the state of the VM.

In many cases the Inspector can determine by context that a particular word value ought to or might point to some specific kind of location. In every case, however, the Inspector investigates the value of the word and determines empirically whether the value points to some known part of the VM state; this is essential for debugging the VM implementation, since those assumptions might not always hold.

Note that this investigation of memory word values can be suspended by turning off the persistent *User Preference* *Investigate memory references*. This does not, however, prevent the value from being read from memory at the conclusion of every refresh cycle.

Color-coding and mouse behavior

When a word value does not point to any known kind of location (for example the parameter boot code end in the Boot Image Inspector), the value is simply displayed in plain hexadecimal (alternate interpretations, for example decimal, are available in a mouseover *Tooltip*). When it does point to contents of a known kind, the display exhibits complex visual and interactive behavior that reveals what is known about the location to which the value refers. This list describes some of those behaviors:

- *Color*: The default display color of a Memory Word Value is black, but if something is learned about where the value points the following color code reveals the kind of data to which the value refers:
  - *green*: points at a *Heap Object*.
  - *blue*: points at a method entry for compiled *Machine Code*.
  - *pale* blue: points into the interior of a method for compiled *Machine Code*.
  - *magenta*: points into *Thread Local Memory*.
  - *red*: points into memory not known to be in a *Memory Region* allocated by the VM.
- *Numeric Display*: A word of bits can be interpreted as several different types of numeric values. For example, floating point register values being displayed in the *Registers Inspector* can be displayed in three different formats: hexadecimal, as a float value, and as a double float value. As noted below, a mouse middle-click over such a Memory Word Value will cause it to cycle among its possible display states.
- *Symbolic Display*: Some values that point to known kinds of information have two modes of display: numeric and symbolic. The default mode depends usually on whether the Inspector assumes from context that a particular value should point to something known. In the Boot Image display, the parameter `boot heap start` is not assumed to point at anything in particular, but the Inspector has discovered that it points at a heap object. On the other hand, the parameter `class registry` is assumed to point to a heap object, so the default display mode is symbolic.

As noted below, a mouse middle-click over such a Memory Word Value will cause it to cycle among its possible display states.

- *Heap Object References*: The symbolic display of a heap object reference (for example, the value of the `class registry` parameter) begins with an integer ID for the object that is unique for the duration of the inspection, followed by the type of the reference, displayed as an unqualified class name. A variant display appears for objects of Maxine's low-level implementation types: `<Maxine role>(<java entity for which the object plays this role>)`. Examples of such roles include Class Actor, Dynamic Hub, and Static Tuple. When a heap object reference is being displayed in numeric mode, symbolic information is among the extra information available as a *Tooltip*, and a mouse left-click will create a new Object Inspector on the object.
- *Machine Code References*: The symbolic display of a word pointing at compiled machine code is displayed (for Java methods) as the unqualified class name, followed by the method name, followed by empty parentheses, followed by a compilation index in square brackets. The compilation index identifies which of the potentially multiple compilations of the method contains the reference location. When a machine code pointer is being displayed in numeric mode, symbolic information is among the extra information available as a *Tooltip*, and a mouse left-click will create a new Method Inspector on the object
- *Left Mouse Button*: A mouse left-click over a Memory Word Value creates an Inspector for what, if anything, is pointed to by the value. If the value is a heap object reference, it creates a new *Object Inspector*. If the value points into machine code, it creates a new Method Inspector displaying the disassembled *Machine Code*.
- *Middle Mouse Button*: A mouse middle-click cycles among the display states of the Memory Word Value under the mouse cursor.
- *Right Mouse Button*: A mouse right-click over a word value causes a menu of commands to be displayed. Some entries in the menu are universal, for example *Copy Word To Clipboard*. When display modes are available, the command *Toggle Display Mode* performs the same function as a mouse middle-click. Commands are available that create a *Memory Inspector* at the location specified by the word value. Yet other commands are sensitive to the particular kind of information pointed to by the value, for example commands associated with Java methods or with Constant Pool entries.

Tooltips

A "tooltip" is a display of a small amount of text that pops up temporarily when the mouse rolls over display element. In the case of Word Value Labels, tooltips display several kinds of useful information that complement the terse displayed text of the element. The duration of each tooltip's appearance can be controlled by the *User Preference ToolTip dismiss*.

Most Word Value Labels in Inspector views appear in tables, which have a "cell" on each row under each column, and in these tables there is a strong convention for what tooltip text appears. The first line of tooltip text usually identifies the particular row under the cursor, and in particular the VM entity that is being portrayed by that row in the table. Examples include:

- `Object header field "MISC"`
- `Instruction 4 "mov"`
- `Thread local "MUTATOR_STATE`

The remaining tooltip lines display additional information, possibly redundant, about the cell under the cursor. For example, a cell in the Value column of an *Object Inspector* or *Memory Inspector* whose memory word contains a `Reference`, displays both the address in hexadecimal and a short description of the referred to object, whereas the table cell itself displays only one of these at a time, depending on its display state. Furthermore, that tooltip also describes the memory region into which the `Reference` points, something that is otherwise visible only by activating a separate column in the view.

Cells in a `Name` column add to the tooltip any of the short "description" strings associated with some Maxine VM internal entities. For example, this string describes the purpose of a VM thread local, and it appears with the tooltip over its name in the *VM Thread Locals Inspector*.

Cells in the `Tag` column of any memory-based Inspector view will display tooltip text (following the line 1 descriptor) that describes

(a) the registers, if any, that point into the row's memory region, (b) the watchpoints, if any, that are set in the row's memory region, and

(c) the watchpoint, if any, that is currently triggered on a location (specified) in the memory region. Some of this information is redundant, since a special cell border reveals the presence of a watchpoint, a special icon and color reveals the location of a triggered watchpoint, and the cell's text lists any registers pointing into the region. Some is not, however, for example the specific address information at which a watchpoint trigger occurred; this information is otherwise only visible in the *Watchpoints Inspector*.

The specific kind of additional information that appears is quite dependent on context: on the particular column (`Value` or `Address`), on an expectation about the value (e.g. "should contain a Reference"), and the actual value discovered in VM memory. This overall approach is designed to offer:

- verbosity and redundancy for the beginner (and sometimes for the pro), and
- additional information for the pro, information that can reduce jumps to another view and reduce the number of columns visible (which in turn frees visual space for other information).

Drag and drop

A Memory Word Value display can also act as the source of a Drag & Drop operation. If the value points into a known region of memory, dragging the value away from the display and dropping it onto the Inspector's background window will produce a *Memory Inspector* whose display begins at that address.

The Memory Inspector

Most of the views provided by the Maxine Inspector display something about the state of the VM that has been read from memory as raw bytes and then been interpreted in useful terms, based on the Inspector's embedded knowledge of the VM's design. Many such views are described in subsequent sections.

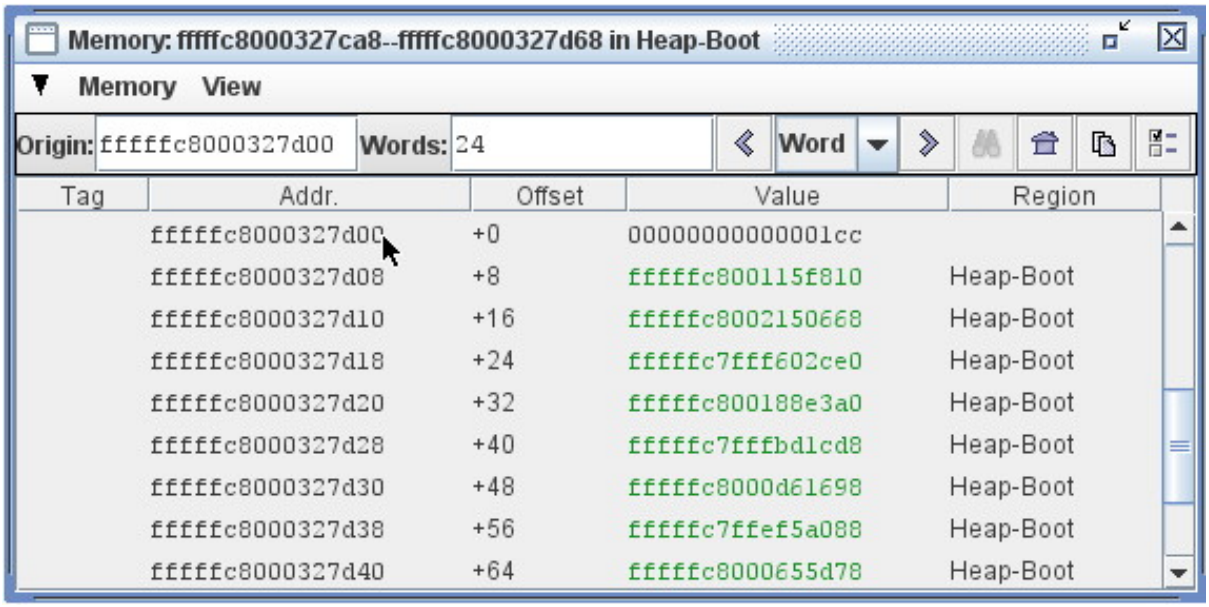
Sometimes, however, it is important to display memory at a very low level, without assumptions about content, and the Maxine Inspector offers low-level views for this purpose.

default "Word" mode

An Inspection session can contain any number of Memory Inspectors. The default behavior of a Memory Inspector is demonstrated by the example to the right. The specified range of memory being displayed appears in the window header, along with the name of the allocated *Memory Region* in which the first

word lies. The memory in the specified range appears, grouped by word, one word per row, using the default columns that appear in the example:

- Tag column: a place where additional information about the memory word can be displayed. For example it displays the names of any registers in the currently selected thread that point at the location. The Tag column also highlights any word where a Watchpoints is set. Many Inspectors have a similar Tag column.
- Addr. column: the location of the first byte in the word, expressed as a hexadecimal memory address.
- Offset column: the location of the word, specified as the number of bytes offset (either positive or negative) from the current origin of the Inspector (more about the origin follows below}.
- Value column: The contents of each word are read from the VM memory each time the VM halts. The values are displayed with numerous visual and interactive behaviors that depend on the value and the context of their appearance. See *Memory Word Values* for details.
- Region column: displays the name of the *Memory Region*, if any, into which the value currently stored in the word points. See *Memory Regions Column*.



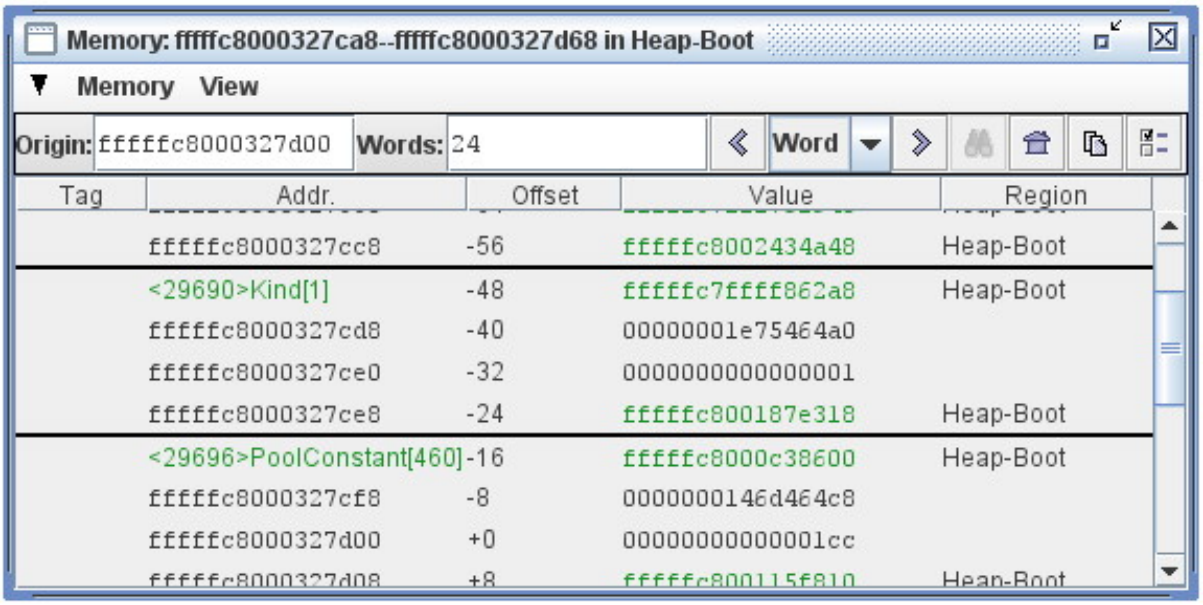
A Memory Inspector can be created in several ways:

- The *Inspect memory at address...* entry in the standard *Memory menu* brings up a dialog in which a starting address for a new Memory Inspector may be entered.
- The *Inspect this object's memory* entry in the Memory menu appearing on any *Object Inspector*.
- Clicking on the *Create cloned copy...* button in the tool bar of any existing Memory Inspector; this creates a new Memory Inspector whose location is identical to the original, but whose subsequent behavior is independent of the original.
- Dragging any *Memory Word Value* to the Inspector's background; if the value can be interpreted as a memory location known to be allocated, a new Memory Inspector will be created started at that location.
- Dragging the display of any *Memory Region* name (for example, any name displayed in a *Memory Region Column*) to the Inspector's background; a new Memory Inspector will be created whose display spans the entire region.

Note that the Memory Inspector depicted in this example is currently in Word mode, as indicated by the pull-down selector in the Inspector's tool bar. In this mode the Back and Forward arrow buttons serve to relocate the viewing region of the Memory Inspector forward or backward one word at a time. The operation of the arrow buttons in other modes (Object and Page modes) is discussed in subsequent sections. Navigation also takes place in response to the scroll bar and by resizing the window.

origin

Every Memory Inspector maintains a current *origin* at all times; this is a word-aligned memory address from which the locations displayed in the `Offset` column are computed. When a Memory Inspector is created, the *origin* is set initially to the first word of the memory being displayed, but the location of the origin is thereafter unconstrained. Commands in the Memory Inspector's View menu, or direct editing of the `Origin` field, allow the origin to be set elsewhere.

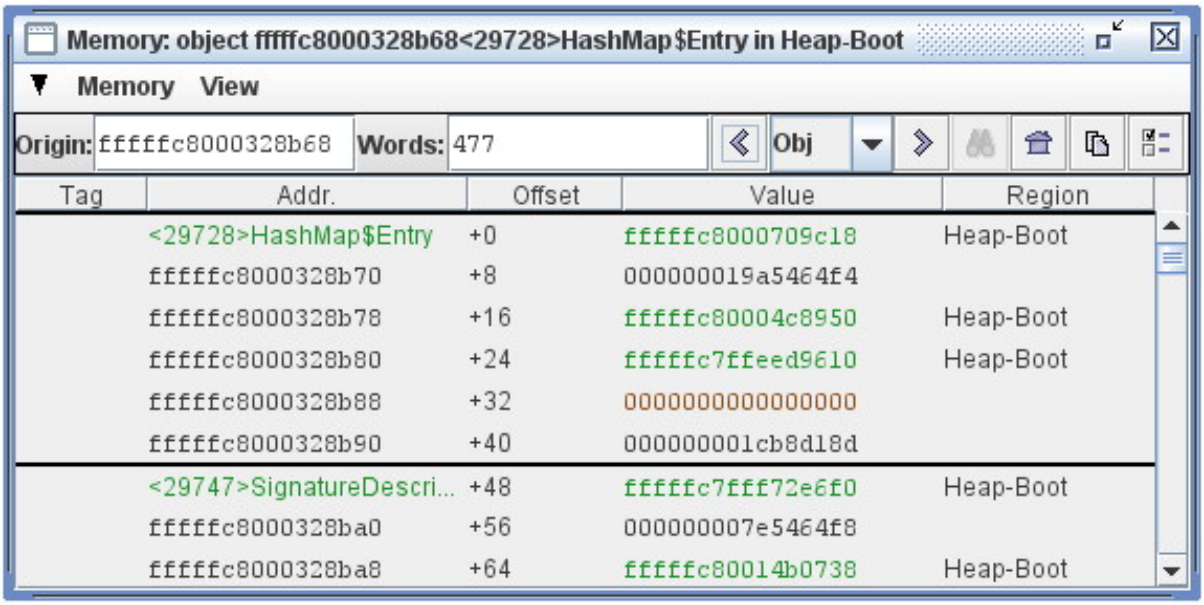


In this example, the displayed memory region is the same as the previous example, but the *origin* has been set to a location in the middle of the displayed region.

This example also shows the graphical separators that are applied by the Memory Inspector whenever it discovers *Heap Object* boundaries in VM memory.

"Object" mode

Navigation in the Memory Inspector is modulated by the mode currently selected via a pull-down selector in the Inspector's tool bar, located between the Back and Forward arrow buttons.

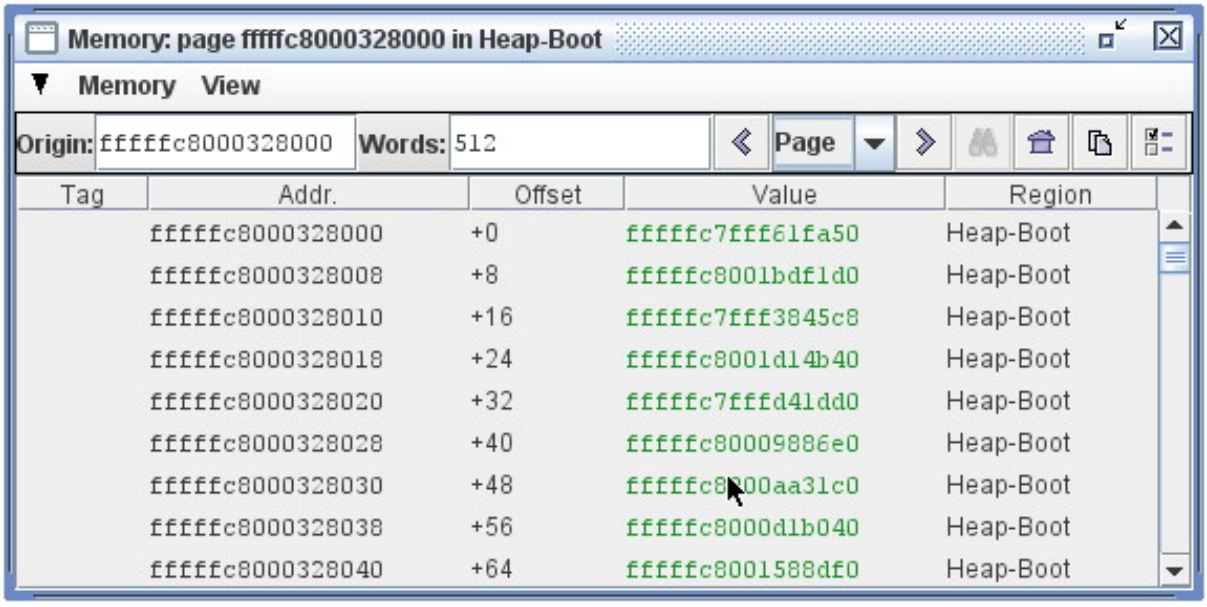


In this example the mode is set to Object, which causes the `Back` and `Forward` buttons to move backward and forward one object at a time, assuming any objects can be located. Each Object-mode move resets the Inspector's origin to the first word of the object's representation and scrolls until that position is in the first viewing position.

These moves do not change the size of the region being displayed, nor do they cause the window to resize around the current object being displayed.

"Page" mode

Navigation in the Memory Inspector is modulated by the mode currently selected via a pull-down selector in the Inspector's tool bar, located between the `Back` and `Forward` arrow buttons.



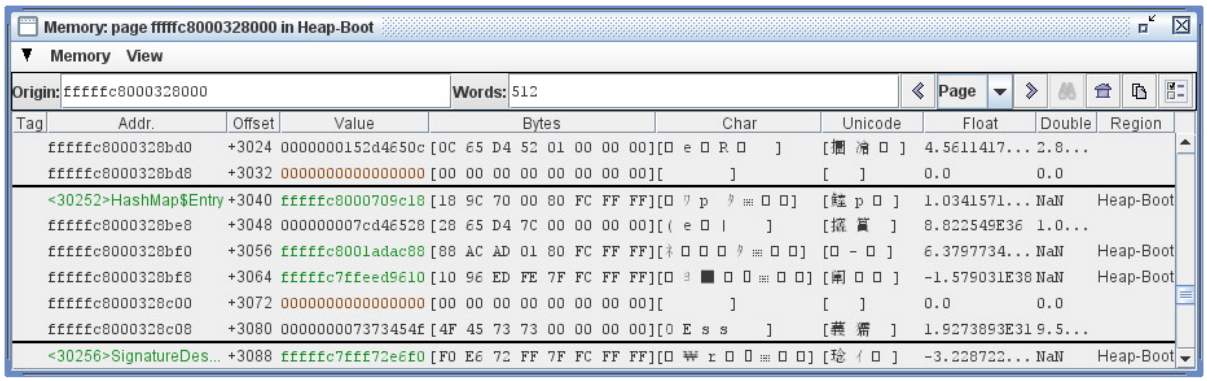
In this example the mode is set to `Page`, which can be very helpful when working on page-based mechanisms in the VM, for example garbage collection. In this mode the size of the region is constrained to equal the page size of the platform, and the origin is constrained to location at page boundaries. Navigation via the `Back` and `Forward` buttons relocates the viewing region by one page per click.

Manually changing either the `Origin` or `Words` size fields causes the mode to revert to Word.

View options

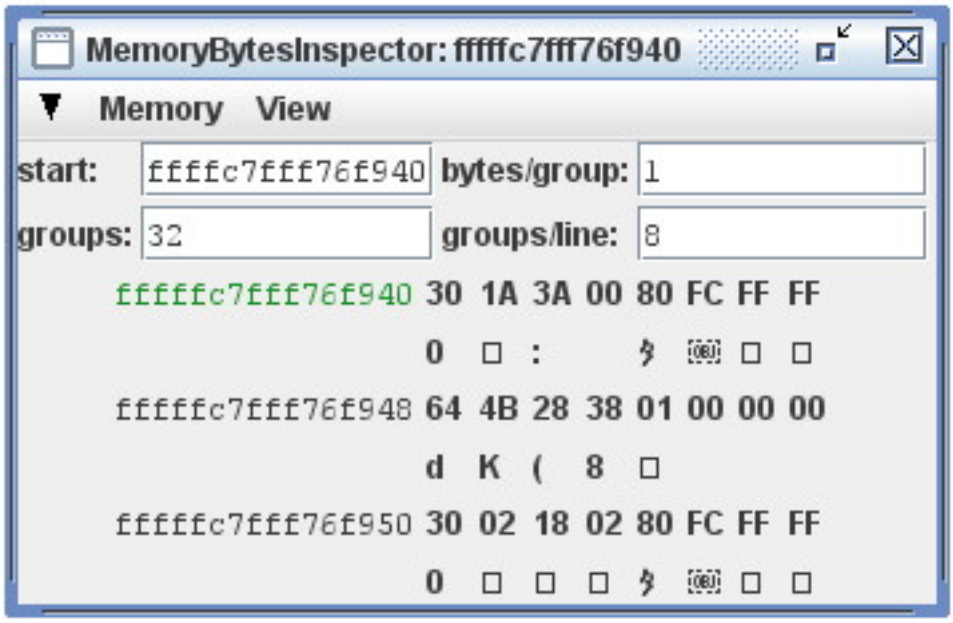
The Memory Inspector provides a number of options for displaying word contents under different interpretations, available via the *View Options* entry in the Inspector's *View* menu. The options dialog can also be invoked by clicking on the rightmost button in the tool bar.

In the example below, all optional columns are displayed. Each column displays the memory contents under a different interpretation: as Bytes, as Chars, as Unicode, as a single-precision Float, and as a Double-precision float.



The Memory Bytes Inspector

There are times when low-level memory inspection in terms of words, the only mode supported by the standard *Memory Inspector* described above, is not flexible enough for the task at hand. In these situations the Memory Bytes Inspector, shown in the example to the right, offers a much more flexible alternative.



This Inspector can be configured to display memory at any location (address, length), and can display memory in any grouping of bytes.

In the special case where bytes appear in groups of 1, as in the example, each byte is also interpreted as an ASCII character. In the special case where bytes appear in groups of 2, each group is also interpreted as a UNICODE character.

In the special case where the address of a byte group is determined empirically to be a valid reference to either a heap object or code, this information can be displayed symbolically. In the example at the right, the Memory Bytes Inspector has observed that the address of the first group points to an object and has color coded the display to indicate that. See *Memory Word Values* for more details.

A Memory Bytes Inspector can be created in several ways:

- The *Inspect memory bytes at address...* entry in the standard Memory menu brings up a dialog in which a starting address for a new Memory Bytes Inspector may be entered.
- The *Inspect memory at Origin as bytes* entry in the View menu appearing on any Memory Inspector.
- A right mouse-click over the Tag column in most memory-based views will produce a popup menu, one entry of which is Inspect this memory as bytes.

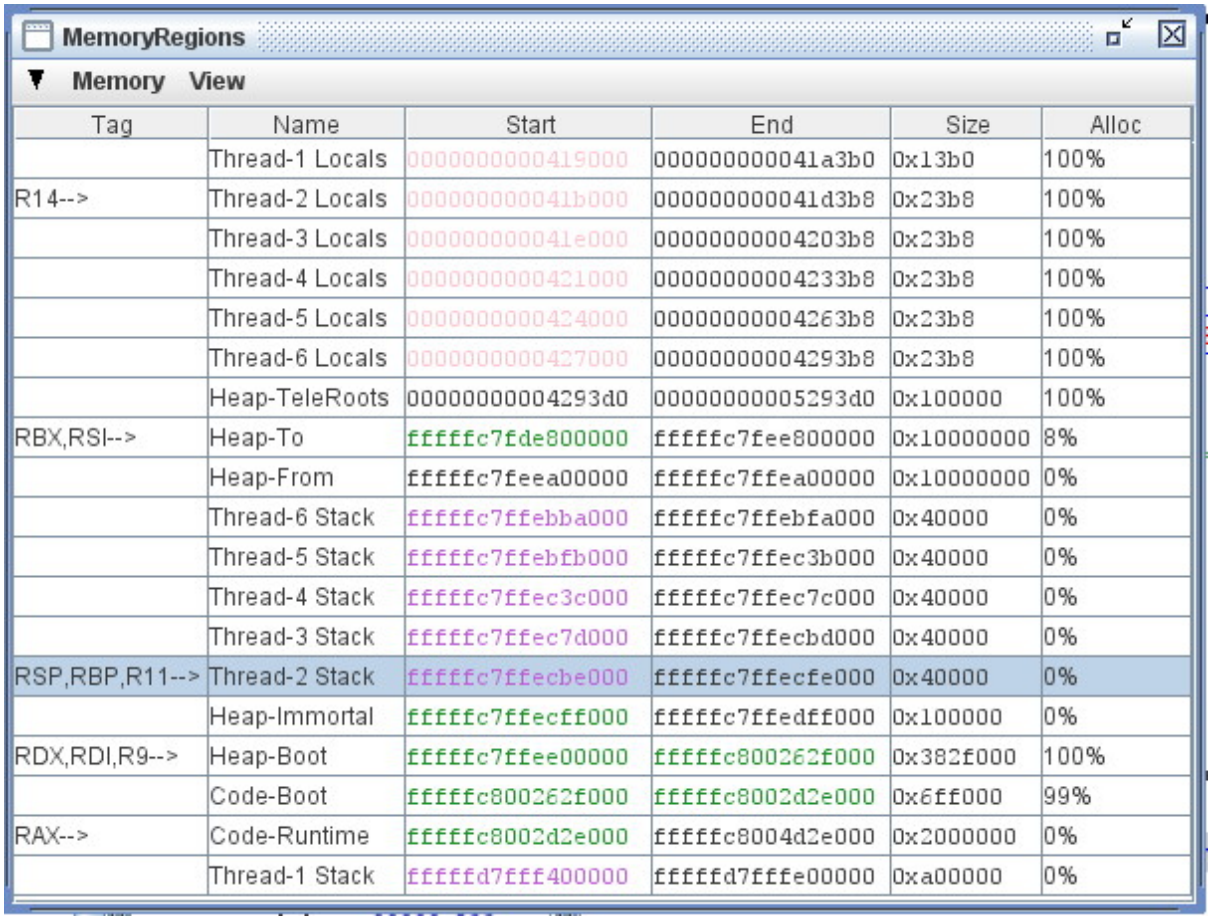
Memory Regions

The Maxine VM allocates memory in regions dedicated to various subsystems. These regions are given names for the purposes of inspection, and the Inspector provides two mechanisms for observing this aspect of of the VM's runtime state: *the Memory Regions Inspector* and *the Memory Regions Column*, which can be optionally displayed with many of the other Inspector Views.

The Memory Regions Inspector

The Memory Regions Inspector displays a tabular summary of every currently allocated region of memory in the running Maxine VM, with the following columns displayed by default:

- *Tag*: a place where additional information about the memory region can be displayed. In the example to the right the Tag entry in the second row notes that register R14 currently points into the region `Thread-2 Locals`. The Tag column also highlights any memory region where one or more *Watchpoints* are set. Many Inspectors have a similar Tag column.
- *Name*: a human readable name that describes its purpose. In the example the two regions named Heap-Boot and Heap-Code are preconfigured as part of the binary boot image (see *Boot Image Inspector*), each in the runtime format of the dynamic heap and code regions respectively. Additional regions are allocated dynamically for code compiled at run time, for example the region named Code-Runtime. Specific heap implementation allocate memory according to a garbage collection scheme, for example the Heap-From and Heap-To regions allocated by a semi-space collector. Finally, a region of memory for the VM's internal *Thread Local Storage* is allocated for each thread, named after the particular thread's ID.
- *Start*, *End*: location of the region, expressed as hexadecimal memory addresses.
- *Size*: number of bytes contained in the memory region, expressed by default in headecimal, but with additional formats available in mouseover Tooltip text.
- *Alloc*: the percentage of the region that has actually been used by the particular subsystem owning the region, if this can be determined.


MemoryRegions

Memory View

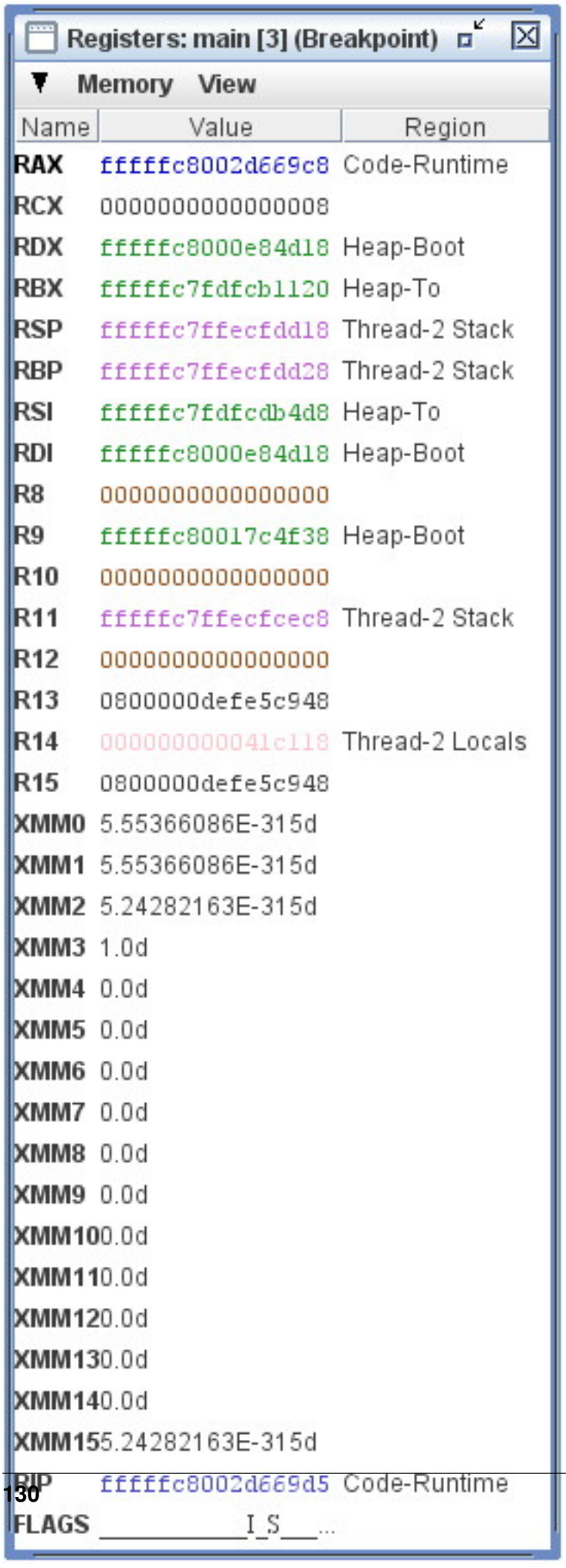
Tag	Name	Start	End	Size	Alloc
	Thread-1 Locals	0000000000419000	000000000041a3b0	0x13b0	100%
R14-->	Thread-2 Locals	000000000041b000	000000000041d3b8	0x23b8	100%
	Thread-3 Locals	000000000041e000	00000000004203b8	0x23b8	100%
	Thread-4 Locals	0000000000421000	00000000004233b8	0x23b8	100%
	Thread-5 Locals	0000000000424000	00000000004263b8	0x23b8	100%
	Thread-6 Locals	0000000000427000	00000000004293b8	0x23b8	100%
	Heap-TeleRoots	00000000004293d0	00000000005293d0	0x100000	100%
RBX,RSI-->	Heap-To	ffffffc7fde800000	ffffffc7fee800000	0x10000000	8%
	Heap-From	ffffffc7feea00000	ffffffc7fffa00000	0x10000000	0%
	Thread-6 Stack	ffffffc7ffebba000	ffffffc7ffebfa000	0x40000	0%
	Thread-5 Stack	ffffffc7ffebfb000	ffffffc7ffec3b000	0x40000	0%
	Thread-4 Stack	ffffffc7ffec3c000	ffffffc7ffec7c000	0x40000	0%
	Thread-3 Stack	ffffffc7ffec7d000	ffffffc7ffecbd000	0x40000	0%
RSP,RBP,R11-->	Thread-2 Stack	ffffffc7ffecbe000	ffffffc7ffecfe000	0x40000	0%
	Heap-Immortal	ffffffc7ffecff000	ffffffc7ffedff000	0x100000	0%
RDX,RDI,R9-->	Heap-Boot	ffffffc7ffee00000	ffffffc800262f000	0x382f000	100%
	Code-Boot	ffffffc800262f000	ffffffc8002d2e000	0x6ff000	99%
RAX-->	Code-Runtime	ffffffc8002d2e000	ffffffc8004d2e000	0x2000000	0%
	Thread-1 Stack	ffffffd7fff400000	ffffffd7fffe00000	0xa00000	0%

In the special case where a *Start* or *End* address is determined empirically by the Inspector to be a valid reference to known kinds of information, this information can be displayed symbolically. In the displayed example, the addresses colored green have been determined to point at heap objects, and the addresses colored magenta have been deteremined to point into thread local storage. Additional behaviors are available at such address display: mouseover Tooltips, mouse left-click, and mouse right-click (all of which are described in more detail in the field values section for *Heap Objects*).

Dragging a hexadecimal address from the *Start* or *End* columns onto the Inspector background causes a *Memory Inspector* to be created starting at that location and having a small default display span. Dragging a name from the *Name* column causes a Memory Inspector to be created whose span is the entire extent of the region.

The Memory Regions Column

Most Inspector views offer multiple columns of display information, only a few of which may be visible by default. The View Options menu item, available in the standard *View menu*, allows user selections of visible columns. This setting is persistent, and it can also be set as a *User Preference*.


Registers: main [3] (Breakpoint)
Memory View
Name Value Region
RAX ffffc8002d669c8 Code-Runtime
RCX 0000000000000008
RDX ffffc8000e84d18 Heap-Boot
RBX ffffc7fdfcb1120 Heap-To
RSP ffffc7ffecfdd18 Thread-2 Stack
RBP ffffc7ffecfdd28 Thread-2 Stack
RSI ffffc7fdfcdb4d8 Heap-To
RDI ffffc8000e84d18 Heap-Boot
R8 0000000000000000
R9 ffffc80017c4f38 Heap-Boot
R10 0000000000000000
R11 ffffc7ffecfcec8 Thread-2 Stack
R12 0000000000000000
R13 0800000defe5c948
R14 00000000004lc118 Thread-2 Locals
R15 0800000defe5c948
XMM0 5.55366086E-315d
XMM1 5.55366086E-315d
XMM2 5.24282163E-315d
XMM3 1.0d
XMM4 0.0d
XMM5 0.0d
XMM6 0.0d
XMM7 0.0d
XMM8 0.0d
XMM9 0.0d
XMM10 0.0d
XMM11 0.0d
XMM12 0.0d
XMM13 0.0d
XMM14 0.0d
XMM15 5.24282163E-315d
RIP ffffc8002d669d5 Code-Runtime
FLAGS _____I_S___...

Every Inspector that display memory values of any kind offers an optional column with the title *Region*. In the example to the right, the *Registers Inspector* is shown with the Memory Regions Column visible.

The Memory Regions Column display is based on a Memory Word Value associated with the particular row. If the Word Value is determined to point to a valid location somewere in the runtime state of the VM, the name of the memory region into which it points is displayed. If the Word Value does not point into a valid memory location, or if it is a different kind of value display, then the The Memory Regions Column is blank.

In the example several of the Word Values point to heap objects: some to objects in the Boot Heap memory region (see *Boot Image Inspector*), and some to the dynamic heap region Heap-To allocated by the semi-space garbage collector, one of several implemented in the Maxine VM. Some values point into the VM's allocation for particular threads. Finally, the RIP register, which is assumed to point into executable code, does indeed point into a compiled method, as shown by the display in symbolic mode.

Dragging a hexadecimal address from the Value column onto the Inspector background causes a Memory Inspector to be created starting at that location and having a small default display span. Dragging a name from the Region column causes a Memory Inspector to be created whose span is the entire extent of the region.

Heap Objects

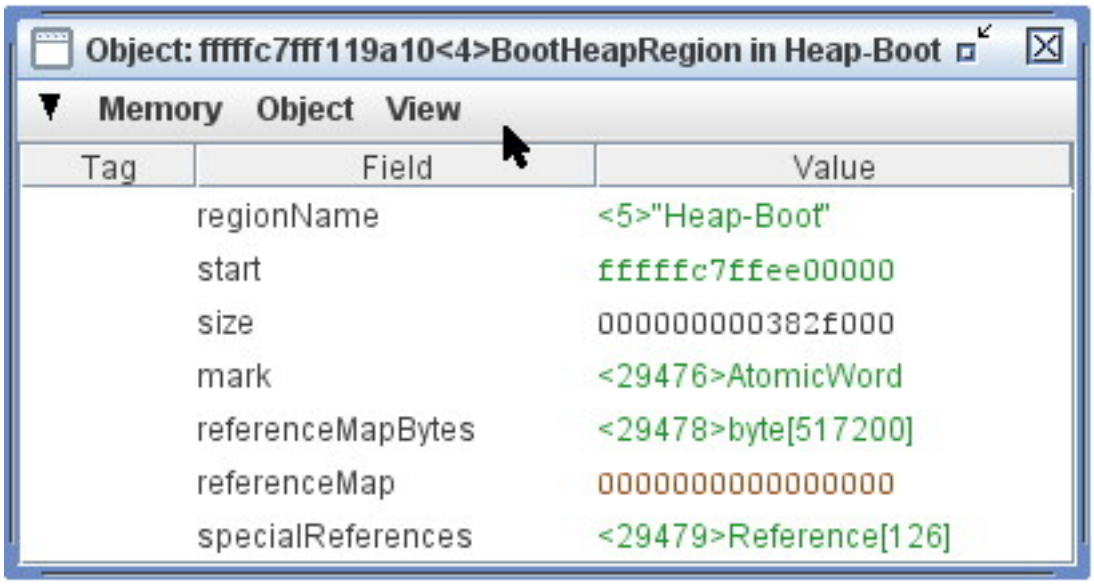
A Maxine Object Inspector displays the contents of a single heap object as a sequence of name/value tuples with additional display options. Variant object representations in the VM are displayed with slightly different kinds of Object Inspectors: tuples (ordinary objects), arrays, and a special hybrid object used in the VM implementation that cannot be expressed as a Java type. Furthermore, certain common types can be displayed in multiple modes, for example the contents of a `char[]` might alternately be displayed as a string.

View a short demo here, or see below for examples and discussion of the heap object inspector's behavior.

Note that the design of Heap Object Inspectors has changed since the demo video. There are many additional display features and options.

Inspecting tuple heap objects

Ordinary objects are referred to as Tuples in the Maxine VM implementation The first example window at the right displays the contents of a simple object of type `com.sun.max.vm.heap.BootHeapRegion`. The object is visualized as a simple list of *Field*/*Value* tuples. In this example, all other view options for the objects are turned off.



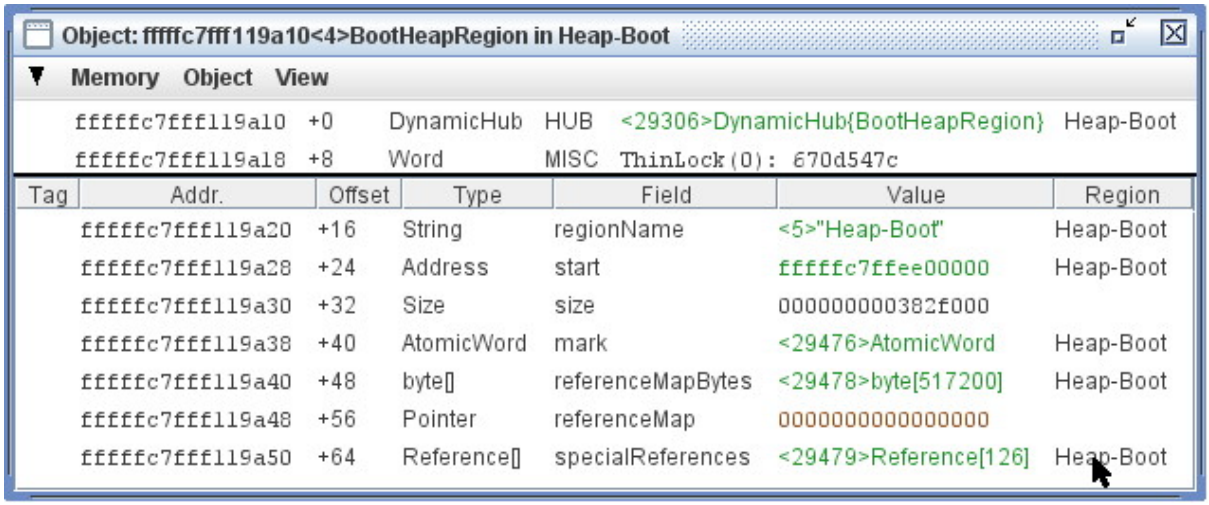
A basic Heap Object Inspector such as this one displays the following elements of a tuple:

- *Title Bar*: The window frame displays a compact string identifying the object: absolute address in memory, an integer ID for the object that is unique for the duration of the inspection, followed by the type of the object (as an unqualified class name) and the *Memory Region* in which it resides.
- *Menu Bar*: The *Standard Menus* relevant to the Object Inspector.
- *Tag* column: The Inspector annotates each field with meta-information that may relate to other aspects of VM state or to the interactive state of the inspection session. For example, an annotation lists the names of all machine *Registers* in the currently selected thread that point at the location represented by the row. A graphical annotations marks the locations of active *Watchpoints* for debugging. A mouse double-left-click in the Tag column toggles on and off the watchpoint at the specified location. A mouse right-click in the Tag column displays a menu of actions relevant to the specific memory location.
- *Field* column: All fields in the object, local or inherited, appear one per row, with the unqualified field name appearing in this column. A mouseover Tooltip reveals the type of the field and the class in which it is declared; both names in the ToolTip are fully qualified.
- *Value* column: The contents of object fields are read from the VM memory each time the VM halts. The values are displayed with numerous visual and interactive behaviors that depend on the value and the context of their appearance. See *Memory Word Values* for details.

Ordinary Java object, such as the one in this example, are represented in the Maxine VM heap as Tuples. There are two other general kinds of objects in the Maxine heap, for which Object Inspector behavior differs somewhat, as described below: *Arrays*, corresponding to ordinary Java arrays, and *Hybrids*, types specialized for the Maxine VM implementation that do not correspond to any Java type.

Object Inspector view options

View options are available available by selecting the View Options entry from the View menu. A dialog permits the request for additional kinds of information, either for the current Object Inspector only or for all subsequently created Object Inspectors. The setting for all subsequently created Object Inspectors is persistent, and it can also be set via the Preferences action (see *User Preferences*). The following example displays a heap object of type BootHeapRegion with all view options turned on.



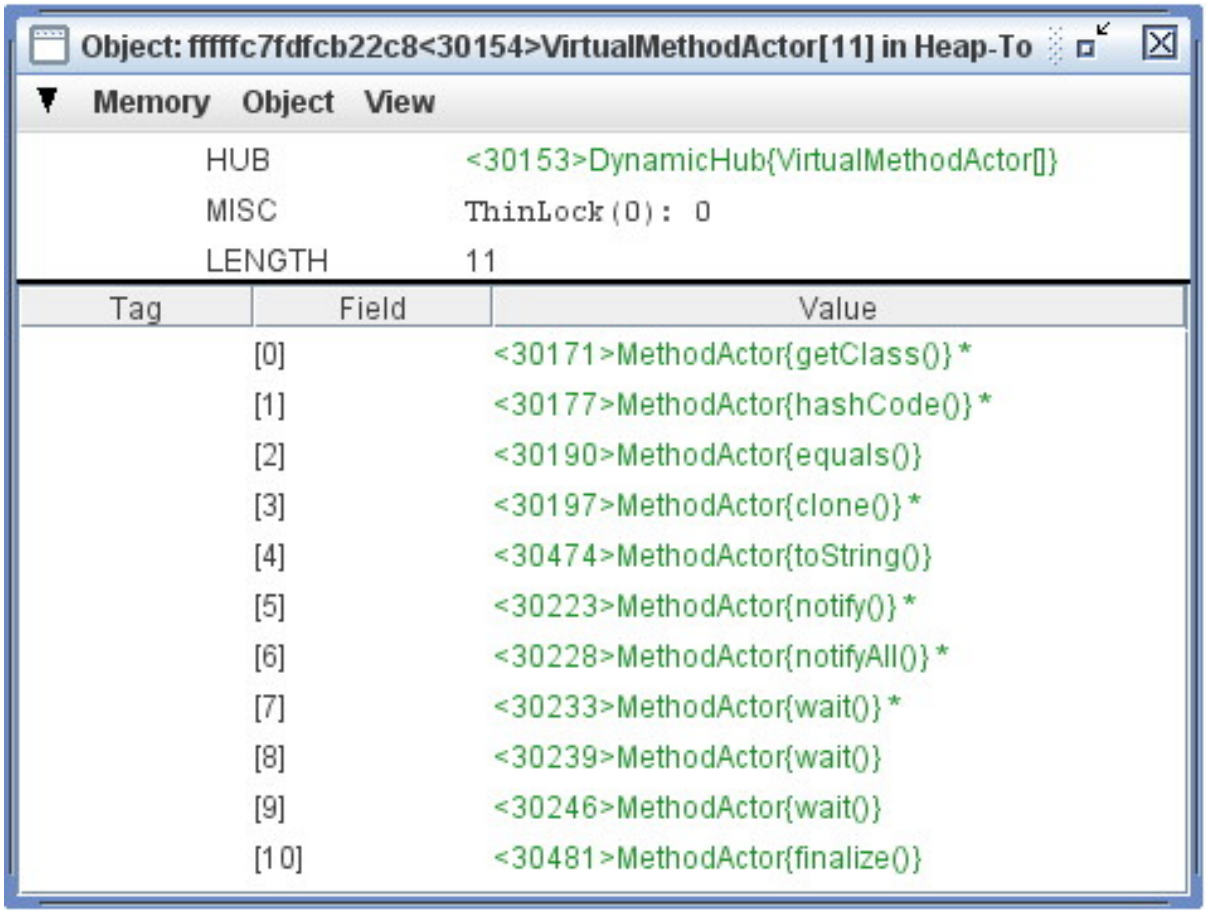
This example displays the same object as the previous example, but with every kind of optional view information enabled. These are listed below, not including the basic display features already described above.

- *Object header*: The Maxine VM implementation of heap objects adds an additional two or three fields in the object representation's header. In the case of simple objects such as this one, the two fields include a reference to the Maxine information (represented as Java objects) concerning the class of the object, followed by a word of bit fields used for a variety of purposes, including locking. A third, when present, specifies the length of the array part of an object (see *Array Objects*) and *Hybrid Objects* below).
- *Addr.* column: displays the absolute current location of the field in VM memory, which may change when the heap is managing by a copying garbage collector. A mouseover Tooltip over this column displays the field's offset from the beginning of the object, the same information displayed in the Offset column. A mouse right-click over an address produces a menu with standard commands for copying the value onto the clipboard and creating a *Memory Inspector* at this location.
- *Offset* column: displays the field's location relative to the origin of the object, where the object layout is determined by a Maxine scheme. In this example, the object layout assigns the origin to memory location 0 in the representation of the object. A moueover Tooltip displays the field's absolute memory location, the same information displayed in the optional Addr. column. A mouse right-click over this column produces a menu with standard commands for copying the value onto the clipboard and creating a *Memory Inspector* at this location.
- *Type* column: displays the Java language type of the value, expressed as either Java primitive type names or unqualified Java class names. Mouseover Tooptips display symbolic information about the Maxine implementation of the type. A mouse right-click produces a menu of commands for inspecting Java objects related to the Maxine implementation of the type.
- *Region* column: displays the name of the *Memory Region*, if any, into which the value currently stored in the field points. See *Memory Regions Column*.

Dragging a hexadecimal address from the Addr. column onto the Inspector background causes a *Memory Inspector* to be created starting at that location and having a small default display span. Dragging a name from the Region column causes a *Memory Inspector* to be created whose span is the entire extent of the region.

Arrays

The Object Inspector displays slightly different information for objects that the VM uses to represent Java arrays, as shown in the example. This Inspector displays an integer array of length 11; a scroll bar would appear when array length exceeds the size of the view window.



Array values are displayed exactly as for field values in ordinary tuple objects: as *Memory Word Values*. In the example, the values are references to objects of type `MethodActor`.

This display differs from an ordinary *Tuple Object Inspector* in two ways. First, the object header contains a third field that holds the length of the array. Second, the Field column identifies the index of each array element.

A mouse double-left-click in the Tag column sets a watchpoint at the specified array element.

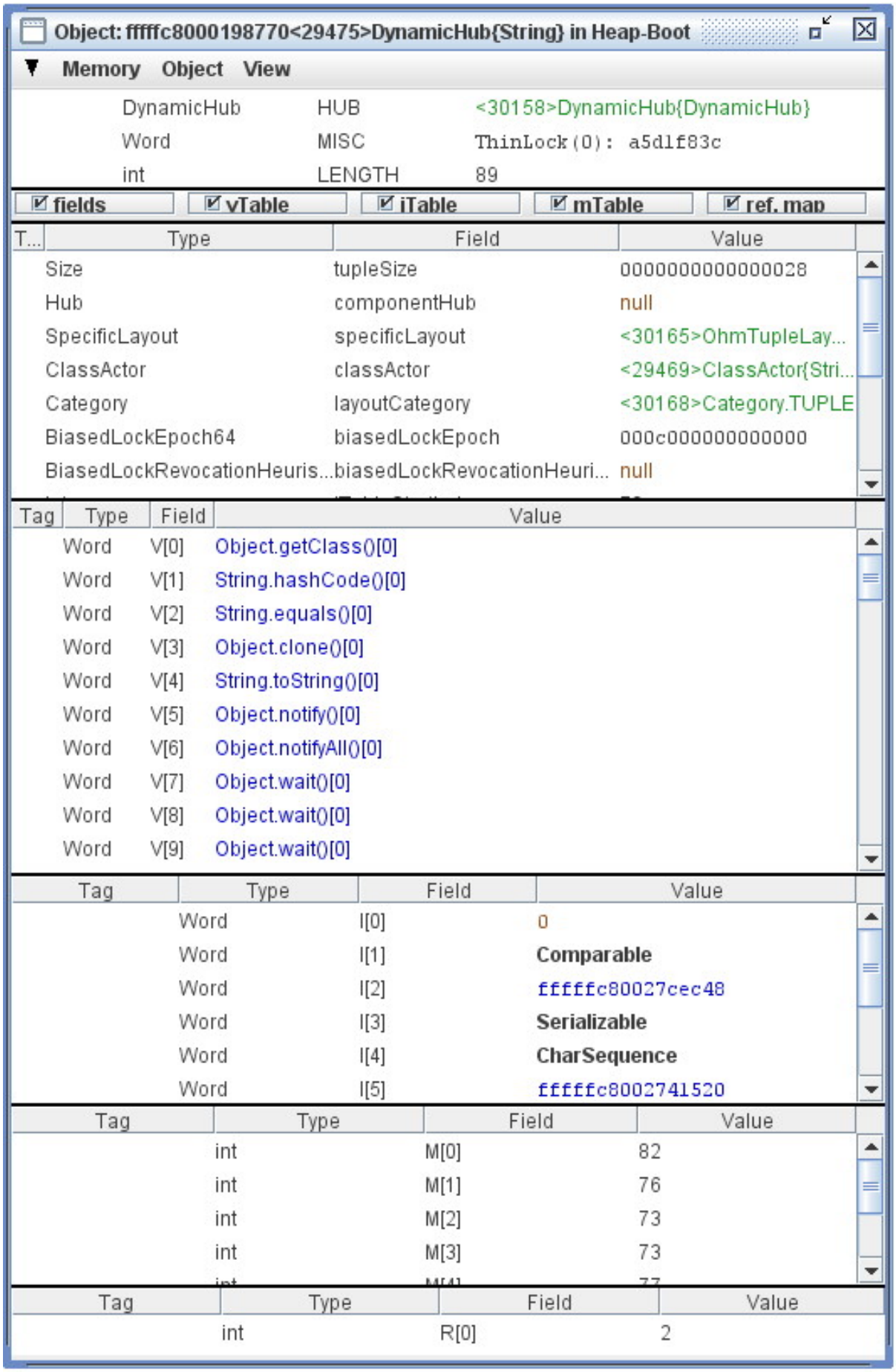
Other than the object header, all view options are turned off in this display. Standard view options are available for *Addr.*, *Offset*, *Type*, and *Region* column. These options are similar to the View Options available for ordinary tuple objects and are available under the View Options entry in the *View menu*.

An additional view option is available for array objects: suppressing the display of `null` elements, where the definition of `null` depends on the particular element type. This can greatly improve visualization of sparsely populated arrays.

Hybrid objects

For performance reasons, the Maxine VM stores much of its class-specific implementation metadata in a special kind of heap object that has no counterpart in the Java language. These objects are *hybrids*: they

contain fields, as with an ordinary tuple object, but they also contain arrays dedicated to implementation data that must be efficiently accessed when examining the representation of an object.


Object: fffffc8000198770<29475>DynamicHub{String} in Heap-Boot
Memory Object View
DynamicHub HUB <30158>DynamicHub{DynamicHub}
Word MISC ThinLock(0): a5d1f83c
int LENGTH 89
fields vTable iTable mTable ref. map
T... Type Field Value
Size tupleSize 0000000000000028
Hub componentHub null
SpecificLayout specificLayout <30165>OhmTupleLay...
ClassActor classActor <29469>ClassActor{Stri...
Category layoutCategory <30168>Category.TUPLE
BiasedLockEpoch64 biasedLockEpoch 000c000000000000
BiasedLockRevocationHeuris... biasedLockRevocationHeuri... null
Tag Type Field Value
Word V[0] Object.getClass()[0]
Word V[1] String.hashCode()[0]
Word V[2] String.equals()[0]
Word V[3] Object.clone()[0]
Word V[4] String.toString()[0]
Word V[5] Object.notify()[0]
Word V[6] Object.notifyAll()[0]
Word V[7] Object.wait()[0]
Word V[8] Object.wait()[0]
Word V[9] Object.wait()[0]
Tag Type Field Value
Word I[0] 0
Word I[1] Comparable
Word I[2] fffffc80027cec48
Word I[3] Serializable
Word I[4] CharSequence
Word I[5] fffffc8002741520
Tag Type Field Value
int M[0] 82
int M[1] 76
int M[2] 73
int M[3] 73
Tag Type Field Value
int R[0] 2

Each ordinary object's header, as shown in earlier examples above, contains a pointer to the object's *Hub*, which is implemented in the heap as a Maxine hybrid object. The example shown to the right is a hybrid object representing the *Dynamic Hub* for objects of type `java.lang.String`. Every `String` object in the heap contains a pointer to this hub.

Each class at runtime also contains static values, represented as an object of the special type StaticTuple, whose metadata is contained in an object of type StaticHub, also represented as a hybrid object.

Note in passing the following circularity: the Hub pointer of a `DynamicHub` points to the `DynamicHub` for class `DynamicHub`.

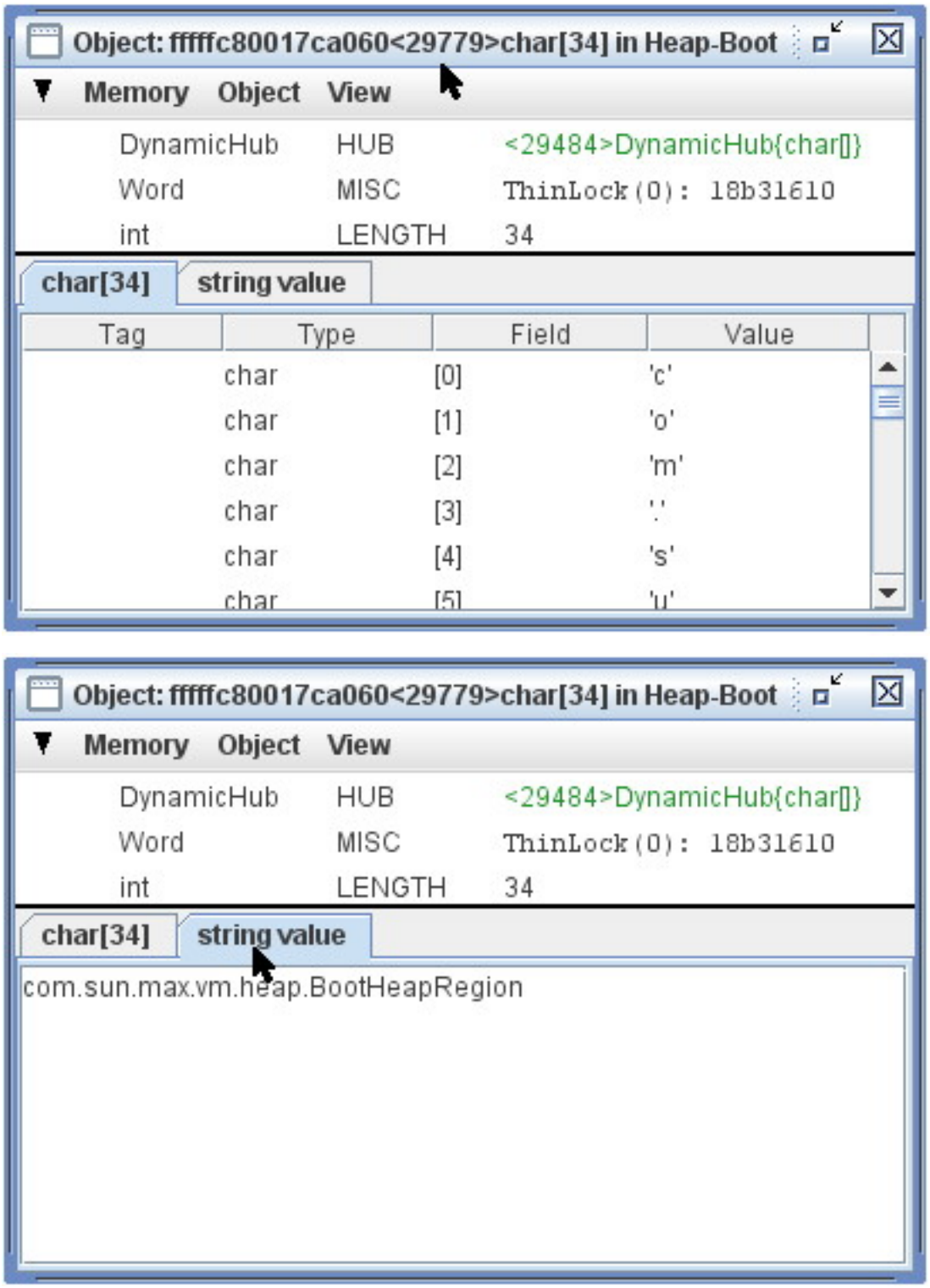
Reflecting the complexity of hybrid objects, the Object Inspector displays a hybrid as a collection of segments, each with different kinds of information.

- As with *Array Objects*, hybrids contain a word in the header that contains the total length of the array part of the object.
- As with *Tuple Objects*, hybrids contain named fields, displayed in the fields segment of the Object Inspector.
- The array segment of a hybrid us used to represent four kinds of information, and the Object Inspector displays each separately: `vTable`, `iTable`, `mTable`, and `Reference Map`. Each array segment behaves as for *Array Objects*.
- Each array segment is individually scrollable, and each can be either displayed or hidden by using checkboxes at the beginning of the Object Inspector.

Specialized Object Inspectors

The Object Inspector can be specialized by adding alternate displays for heap objects of particular types. Several are currently in place, most of which display a textual summary of the object's contents.

In two examples shown, a char array is shown to have such a specialized alternate configured, evident by the appearance of window tabs that select the display. The standard array display appears in the upper example, while the textual summary appears in the lower example.



Object view canonicalization

In ordinary operation, the Inspector creates at most one Object Inspector per unique object in the VM's heap. A user request to view an object, for example by clicking on a value field that points at an object (see [Memory Word Values|Inspector-Memory Word Values), will cause a new Object Inspector to be created only if one does not already exist; if one does exist, it is simply brought forward and into full view. The determination is made by comparing the memory location of the two potentially identical

objects.

In some situations, however, especially during garbage collection, the Inspector may not be able to make this determination identity correctly. For example, a relocating garbage collector may create a copy of an object's representation, and this relationship may not be detectable immediately. The Inspector is designed to sort this out as much as possible, most importantly by stopping the VM at the conclusion of each GC cycle and reviewing reviewing for duplications its table of VM heap objects.

This is work in progress, and the Inspector may not always get identity sorted out correctly in every situation for every implementation of garbage collection.

Machine Code

A Maxine Method Inspector displays code associated with a method body in several ways. Here we show how machine code can be disassembled and displayed with useful interactive behavior.

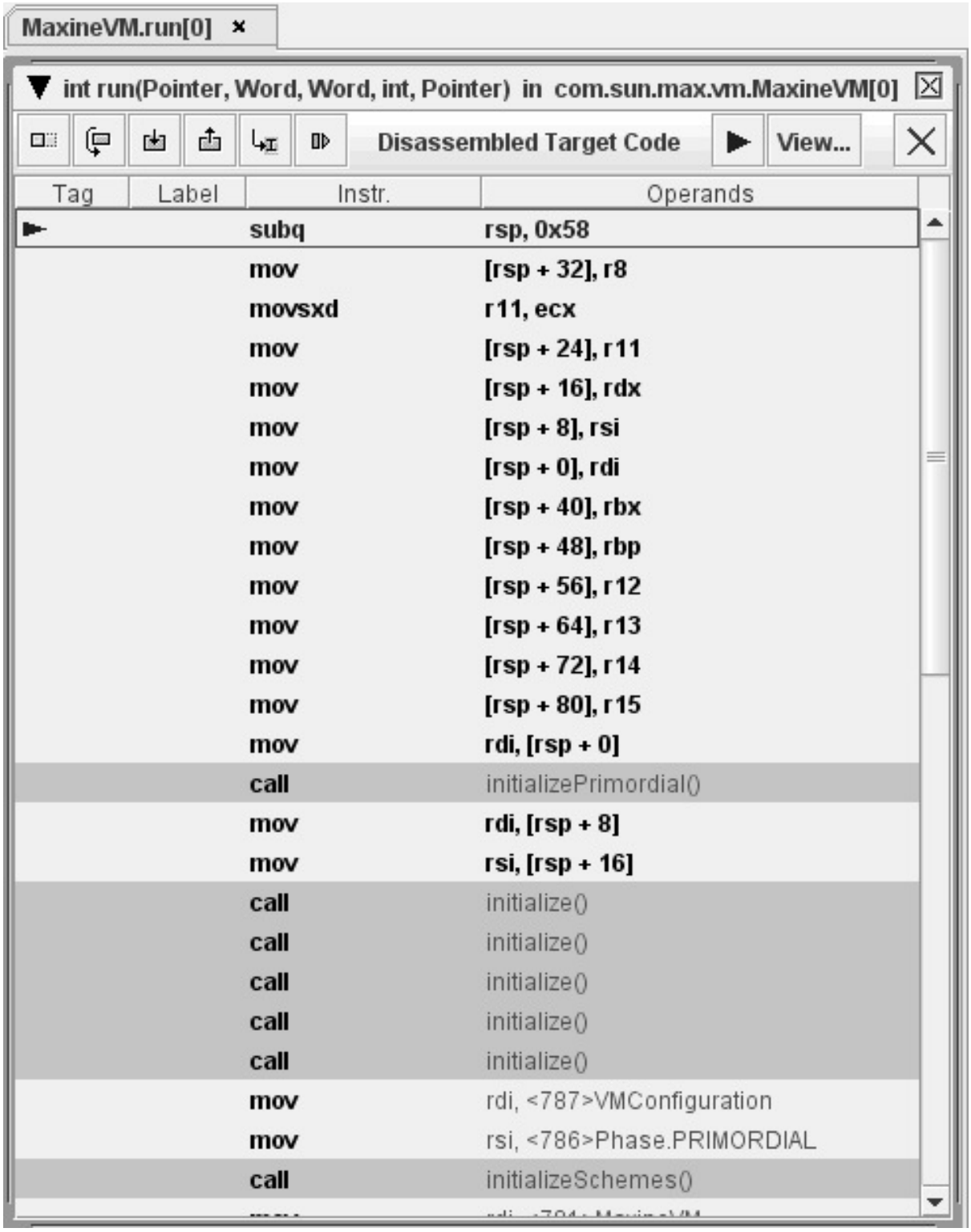
View a short demo here, or see below for a discussion and screen snapshots of the Machine Code Inspector

Note that the design of Method Inspectors has changed somewhat since the demo video.

Method Inspector with machine code

The first example shows a Method Inspector displaying the disassembled machine code for the Java method `com.sun.max.vm.MaxineVM.run()`, which is called by the VM at the conclusion of the startup sequence. Display features include:

- A *tab* that distinguishes the method inspector from others in a "tabbed window";
- A *window header* that identifies the method in detail (which information is also available on the tab as a mouseover Tooltip);
- A *suffix* to the name that identifies the specific compilation of the method; in the example the suffix "0" identifies the machine code as the first entry in the method's compilation history;
- A number of *command buttons* for Debugging;
- A dialog for setting *view options*, available from the *View...* button, in which specific display columns can be selected or deselected (the next example shows all columns);
- A *Tag* column that displays markers related to Debugging, such as the triangular symbol for the current Instruction Pointer in the first row of the example;
- A *Label* column that displays symbolic labels generated by the disassembler; information about the actual location in memory is available as a mouseover Tooltip in this column, and a menu of commands related memory locations is available via mouse right-click over this column;
- An *Instruction* column displaying mnemonic machine operations, as configured for the target instruction set; and
- An *Operands* column displaying mnemonics for machine code operands, as configured for the platform instruction set; in the special case where memory addresses appear in machine code operands, the inspector empirically determines whether the address points at a heap object or code entry, and if so, displays that information symbolically; additional display and interactive options are available over such references, as described in the Field Values section in *Heap Objects*.

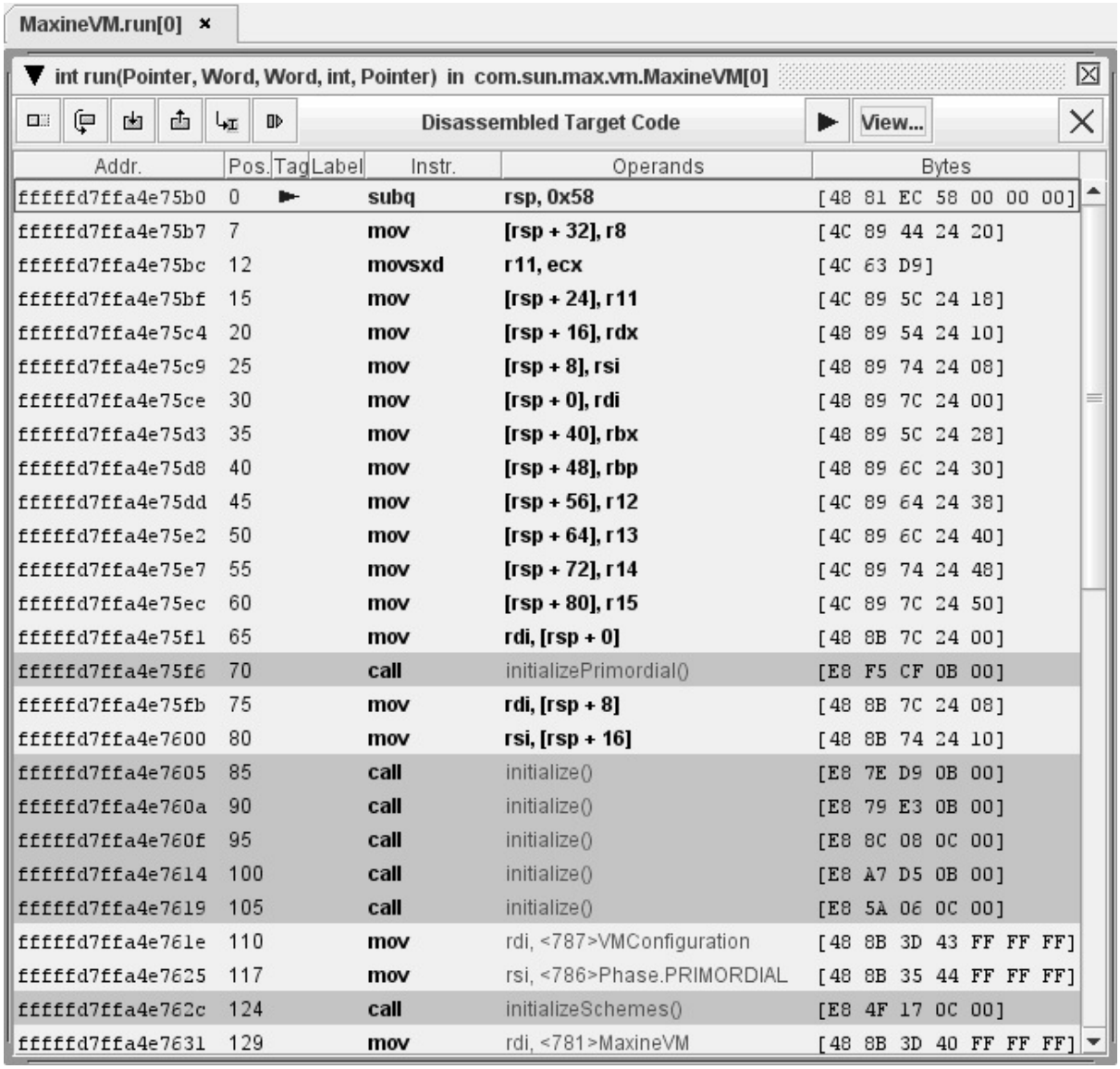


Optional display columns

The second example window shows the same method inspection as the first, but with all possible columns selected for view. The addtional columns include:

- An *Address* column displaying absolute memory location of the code, which information is also available via mouseover Tooltip on the *Label* column;

- A *Position* column displaying memory location as a byte position relative to the beginning of the method, which information is also available via mouseover Tooltip on the *Label* column; and
- A *Bytes* column that displays each instruction in raw bytes.



Addr.	Pos.	Tag	Label	Instr.	Operands	Bytes
ffffffd7ffa4e75b0	0	►		subq	rsp, 0x58	[48 81 EC 58 00 00 00]
ffffffd7ffa4e75b7	7			mov	[rsp + 32], r8	[4C 89 44 24 20]
ffffffd7ffa4e75bc	12			movsxd	r11, ecx	[4C 63 D9]
ffffffd7ffa4e75bf	15			mov	[rsp + 24], r11	[4C 89 5C 24 18]
ffffffd7ffa4e75c4	20			mov	[rsp + 16], rdx	[48 89 54 24 10]
ffffffd7ffa4e75c9	25			mov	[rsp + 8], rsi	[48 89 74 24 08]
ffffffd7ffa4e75ce	30			mov	[rsp + 0], rdi	[48 89 7C 24 00]
ffffffd7ffa4e75d3	35			mov	[rsp + 40], rbx	[48 89 5C 24 28]
ffffffd7ffa4e75d8	40			mov	[rsp + 48], rbp	[48 89 6C 24 30]
ffffffd7ffa4e75dd	45			mov	[rsp + 56], r12	[4C 89 64 24 38]
ffffffd7ffa4e75e2	50			mov	[rsp + 64], r13	[4C 89 6C 24 40]
ffffffd7ffa4e75e7	55			mov	[rsp + 72], r14	[4C 89 74 24 48]
ffffffd7ffa4e75ec	60			mov	[rsp + 80], r15	[4C 89 7C 24 50]
ffffffd7ffa4e75f1	65			mov	rdi, [rsp + 0]	[48 8B 7C 24 00]
ffffffd7ffa4e75f6	70			call	initializePrimordial()	[E8 F5 CF 0B 00]
ffffffd7ffa4e75fb	75			mov	rdi, [rsp + 8]	[48 8B 7C 24 08]
ffffffd7ffa4e7600	80			mov	rsi, [rsp + 16]	[48 8B 74 24 10]
ffffffd7ffa4e7605	85			call	initialize()	[E8 7E D9 0B 00]
ffffffd7ffa4e760a	90			call	initialize()	[E8 79 E3 0B 00]
ffffffd7ffa4e760f	95			call	initialize()	[E8 8C 08 0C 00]
ffffffd7ffa4e7614	100			call	initialize()	[E8 A7 D5 0B 00]
ffffffd7ffa4e7619	105			call	initialize()	[E8 5A 06 0C 00]
ffffffd7ffa4e761e	110			mov	rdi, <787>VMConfiguration	[48 8B 3D 43 FF FF FF]
ffffffd7ffa4e7625	117			mov	rsi, <786>Phase.PRIMORDIAL	[48 8B 35 44 FF FF FF]
ffffffd7ffa4e762c	124			call	initializeSchemes()	[E8 4F 17 0C 00]
ffffffd7ffa4e7631	129			mov	rdi, <781>MaxineVM	[48 8B 3D 40 FF FF FF]

Bytecode

A Maxine Method Inspector displays the code in a method body in several ways. Here we show how Java bytecode can be disassembled and displayed with useful interactive behavior.

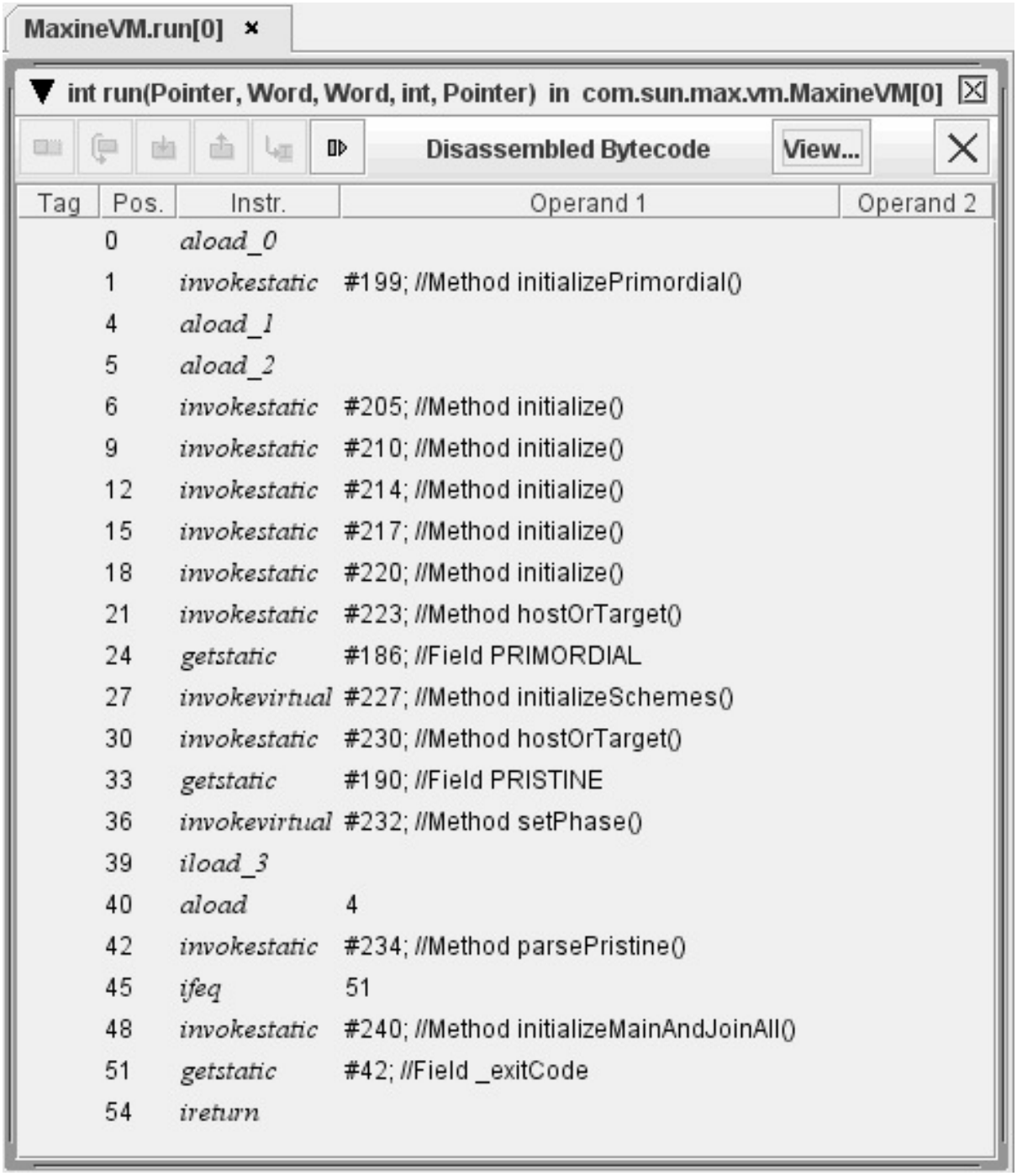
View a short demo here, or see below for a discussion and screen snapshots of the Bytecode Inspector.

Note that the design of Method Inspectors has changed somewhat since the demo video.

Method Inspector with bytecode

The first example shows how the Method Inspector displays disassembled bytecodes for the Java method `com.sun.max.vm.MaxineVM.run()`, which is called by the VM at the conclusion of the startup sequence. Display features include:

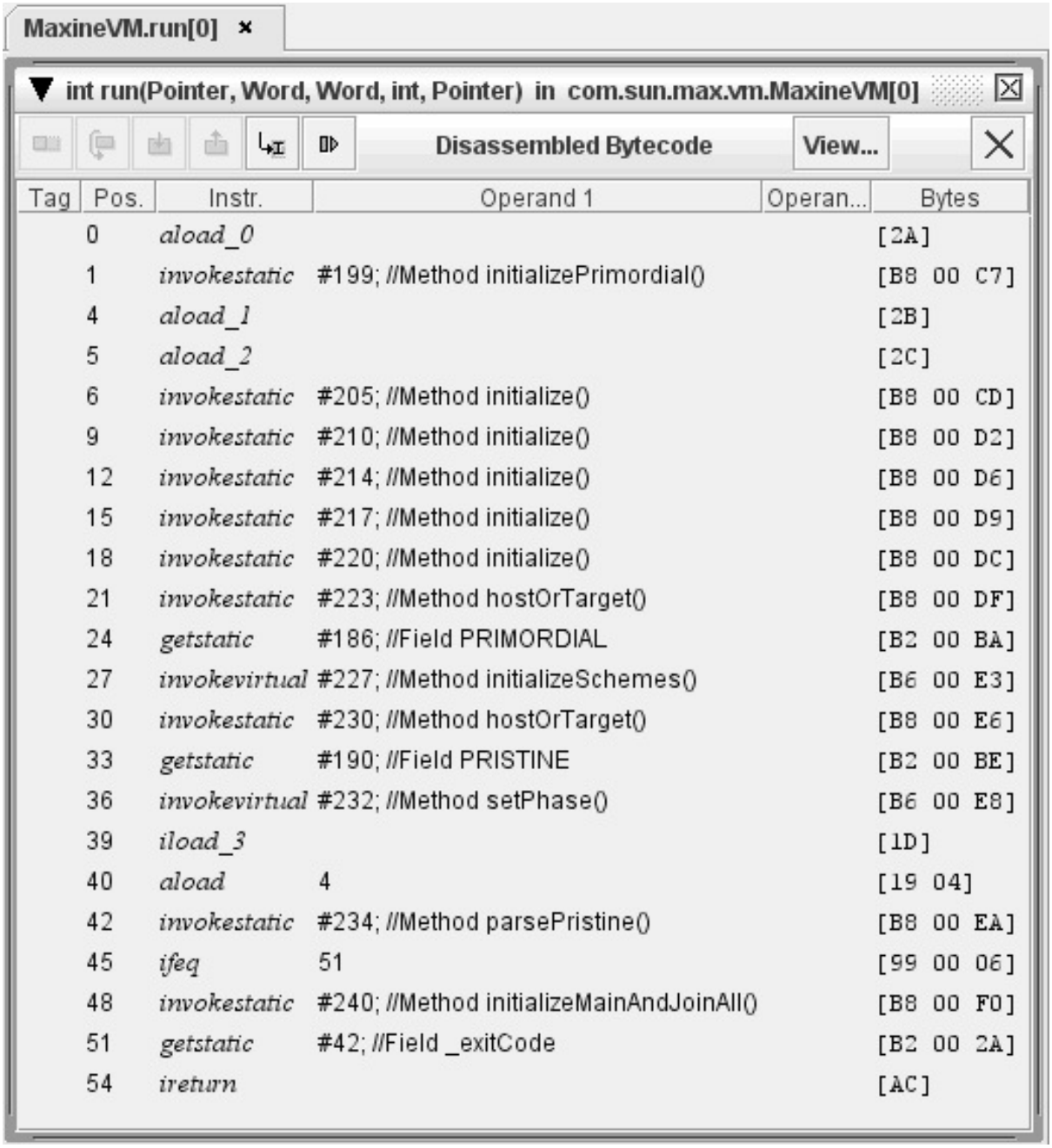
- A *tab* that distinguishes the method inspector from others in a "tabbed window";
- A *window header* that identifies the method in detail (which information is also available on the tab as a mouseover Tooltip);
- A number of *command buttons* for debugging;
- A dialog for setting *view options*, available from the *View...* button, in which specific display columns can be selected or deselected (the next example shows all columns);
- A *Tag* column that displays markers related to debugging, such as instruction pointer, call return site, and breakpoint;
- A *Position* column that displays the byte offset at the beginning of each instruction, relative to the beginning of the code block;
- An *Instruction* column displaying the mnemonic name of each bytecode instruction, as defined by the specification for the Java Virtual Machine; and
- Two Operand columns displaying bytecode operands in a format based loosely on the examples in the book The Java Virtual Machine Specification and on the output of the command line disassembler javap; bytecode operands identifying constant pool entries that reference Java language objects are displayed symbolically, and the displays have useful display and interactive behavior; for example, a mouseover Tooltip displays the full Java description for the reference and identifies whether the reference has been resolved.



Optional display columns

The second example window shows the same method inspection as above, but with an additional column selected for view:

- A Bytes column displays each instruction in raw bytes.



Tag	Pos.	Instr.	Operand 1	Operan...	Bytes
	0	*aload_0*			[2A]
	1	*invokestatic*	#199; //Method initializePrimordial()		[B8 00 C7]
	4	*aload_1*			[2B]
	5	*aload_2*			[2C]
	6	*invokestatic*	#205; //Method initialize()		[B8 00 CD]
	9	*invokestatic*	#210; //Method initialize()		[B8 00 D2]
	12	*invokestatic*	#214; //Method initialize()		[B8 00 D6]
	15	*invokestatic*	#217; //Method initialize()		[B8 00 D9]
	18	*invokestatic*	#220; //Method initialize()		[B8 00 DC]
	21	*invokestatic*	#223; //Method hostOrTarget()		[B8 00 DF]
	24	*getstatic*	#186; //Field PRIMORDIAL		[B2 00 BA]
	27	*invokevirtual*	#227; //Method initializeSchemes()		[B6 00 E3]
	30	*invokestatic*	#230; //Method hostOrTarget()		[B8 00 E6]
	33	*getstatic*	#190; //Field PRISTINE		[B2 00 BE]
	36	*invokevirtual*	#232; //Method setPhase()		[B6 00 E8]
	39	*iload_3*			[1D]
	40	*aload*	4		[19 04]
	42	*invokestatic*	#234; //Method parsePristine()		[B8 00 EA]
	45	*ifeq*	51		[99 00 06]
	48	*invokestatic*	#240; //Method initializeMainAndJoinAll()		[B8 00 F0]
	51	*getstatic*	#42; //Field _exitCode		[B2 00 2A]
	54	*ireturn*			[AC]

Multi-code Method Views

A Maxine Method Inspector displays the code in a method body in several ways, and can do so in more than one way simultaneously. Here we show how machine code and bytecode for a method body can be viewed together.

View a short demo here, or see below for a discussion of the combined method views.

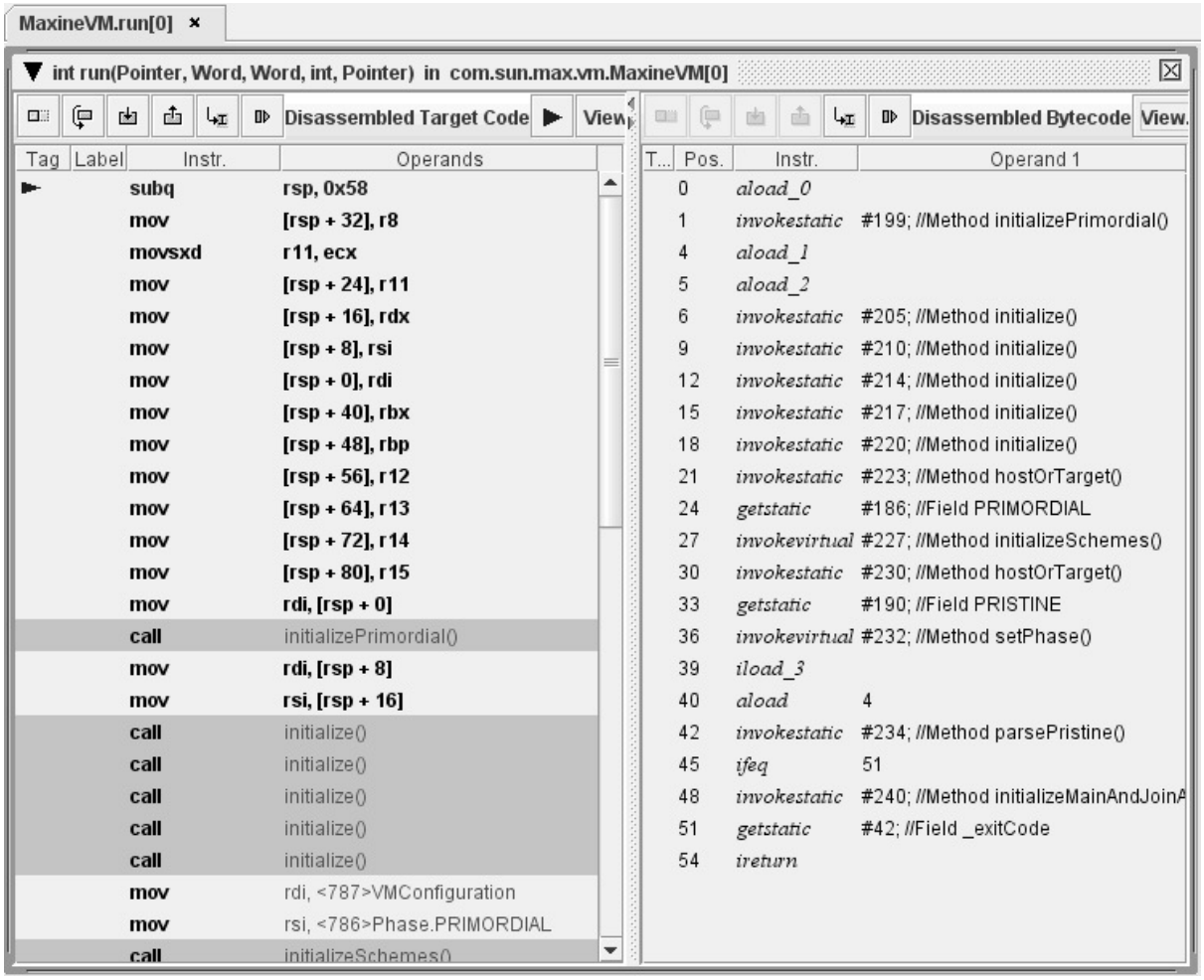
Note that the design of Method Inspectors has changed somewhat since the demo video.

Method Inspector with machine code and bytecode

The example below shows the same method used in previous examples: Java method `com.sun.max.vm.MaxineVM.run()`, which is called by the VM at the conclusion of the startup sequence. In this view, both machine code and bytecode have been enabled, managed via the menu available on the triangle at the upper left corner of the Method Inspector.

It is possible to debug in a multi-code method view. When a reliable map between machine code and bytecode locations is available (currently true only for the Maxine VM's template-based JIT compilations), the Instruction Pointer location will be visible in both views and will track correctly during single stepping. *Breakpoints* can be set in either code view, although the detailed behavior of the breakpoints may differ in some situations.

[STRIKEOUT:A third option, to display source code, will be added.]



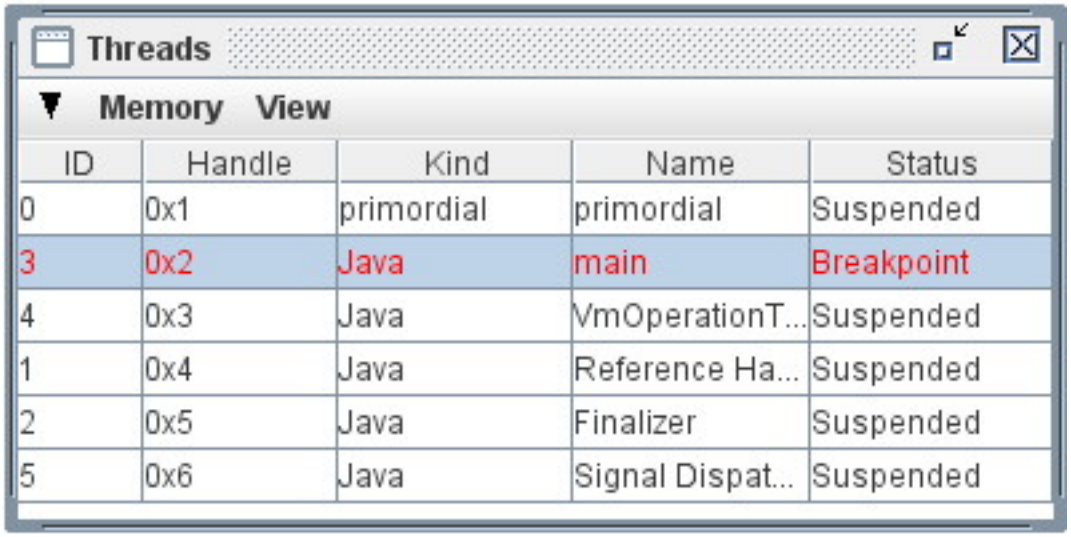
Threads

The Threads Inspector is one of the Maxine Inspector's tools for examining machine state during execution of the Maxine VM. It also serves to change the focus of other, thread-specific views in the Inspector: *Thread Locals*, *Registers*, and *Stacks*.

View a short demo here, or see below for a discussion and screen snapshot of the Threads Inspector.

Note that the design of the Thread Inspector has changed somewhat since the demo video.

The Threads Inspector



The Threads Inspector displays a table of basic information about each thread that exists in the VM process, including by default the following columns:

- *ID*: a numeric identifier associated with the native thread in the underlying OS.
- *Handle*: a numeric identifier associated with Java threads managed by the Maxine VM.
- *Kind*: a string identifying the kind of thread, for example "Java" for threads created and managed by the VM, "primordial" for the the special native thread used to bootstrap the VM, and no name for other native threads.
- *Name*: a human-readable string assigned by the VM to describe the role of the thread, for example those showing in the window include the main Java thread, Java utility threads for reference management and finalization, and the special native thread used to bootstrap the VM which we call the "primordial" thread.
- *Status*: describes what is known about the state of the thread, for example "Suspended" or at "Breakpoint".

Aside: There is no thread in a Maxine VM that either runs or supports the Maxine Inspector, a crucially important design decision for enabling the debugging of low level VM mechanisms. The Inspector runs in a separate process and communicates with the VM process in an OS-specific fashion, for example via "libproc" in Solaris.

The current thread selection

A mouse left-click on one of the rows causes the displayed thread to become the "current thread selection" shared by all tools in the Inspector (see *User Focus*). Several Inspector views display thread-specific information, based on the current thread selection: the *Thread Locals Inspector*, the *Registers Inspector*, and the *Stack Inspector*. Furthermore, most memory-based views contain a *Tag* column in which each row may contain the name of any registers for the currently selected thread that point into the memory designated by the row.

Thread Locals

The Maxine VM allocates an internal Memory Region for each Thread that is used to store implementation data that is private (or "local") to the threads implementation. This storage is not to be confused with thread-local storage provided as part of the Java programming model.


Thread Local Variables: main [3] (Breakpoint)
Memory View
TRIGGERED ENABLED DISABLED
start: 00000000041bff8 end: 00000000041c110 size: 280
T... Pos. Field Value
+0 SAFEPOINT_LATCH void
+8 SAFEPOINTS_ENABLED_THREAD_LO... 00000000041c118
+16 SAFEPOINTS_DISABLED_THREAD_LO... 00000000041c238
+24 SAFEPOINTS_TRIGGERED_THREAD_L... 00000000041bff8
+32 NATIVE_THREAD_LOCALS 00000000041c358
+40 FORWARD_LINK 0000000000000000
+48 BACKWARD_LINK 0000000000425118
+56 VM_OPERATION null
+64 EXCEPTION_OBJECT null
+72 ID 0000000000000003
+80 VM_THREAD <29833>VmThread{mai...
+88 JNI_ENV ffffffd7fff1c2010
+96 LAST_JAVA_FRAME_ANCHOR 0000000000000000
+1... MUTATOR_STATE 0000000000000000
+1... FROZEN 0000000000000000
+1... TRAP_NUMBER 0000000000000000
+1... TRAP_INSTRUCTION_POINTER 0000000000000000
+1... TRAP_FAULT_ADDRESS 0000000000000000
+1... TRAP_LATCH_REGISTER 0000000000000000
+1... HIGHEST_STACK_SLOT_ADDRESS ffffffc7ffecfe000
+1... LOWEST_STACK_SLOT_ADDRESS ffffffc7ffecbf000
+1... LOWEST_ACTIVE_STACK_SLOT_ADD... 0000000000000000
+1... STACK_REFERENCE_MAP 00000000041c3a8
+1... STACK_REFERENCE_SIZE 0000000000001008
+1... IMMORTAL_ALLOCATION_ENABLED 0000000000000000
+2... INTERPRETED_METHOD null
+2... NATIVE_CALL_STACK_SIZE 0000000000000000
+2... TLAB_TOP 0000000000000000
+2... TLAB_MARK 0000000000000000
+2... TLAB_TOP_TMP 0000000000000000

The VM's Thread Local Variables for the currently selected thread (see *Threads*) are displayed by the Thread Locals Inspector, as shown in the example to the right. These variables are part of each thread's internal implementation, in the form of word-length name-value pairs described by default with the following columns:

- *Tag*: as with other memory-related views, the *Tag* column displays the names of any registers for the currently selected thread that currently point at the row's memory location, as well as the possible presence of a *Watchpoint*. A mouse double-left-click in the Tag column sets a watchpoint at the specified location.
- *Pos.*: the offset of the local variable slot from the beginning of the variable set, specified in bytes.
- *Field*: the name by which the VM's internal implementation knows the particular thread-local variable. A mouseover Tooltip displays a documentation string, specified in the VM's implementation, that describes the role of the variable in human-readable form.
- *Value*: the current contents of each word using techniques described elsewhere (see *Memory Word Values*).

The selection of visible columns can be selected using a dialog created by the View Options entry in the Inspectors View menu. Additional columns available include Address and the standard *Memory Region Column*.

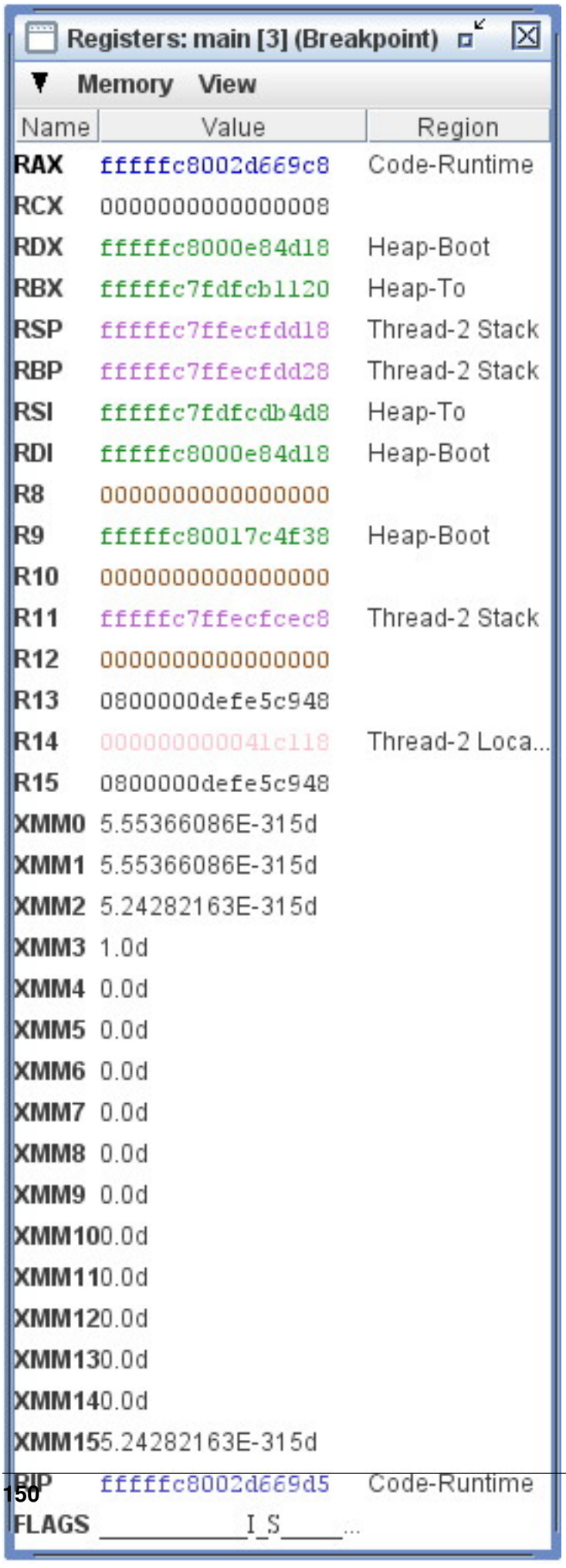
Note that the Maxine VM implementation maintains three copies of the thread locals, identified by the three tabs that select which one to view: `TRIGGERED`, `ENABLED`, or `DISABLED`.

Registers

The Registers Inspector is one of the Maxine Inspector's tools for examining machine state during execution of the Maxine VM.

View a short demo here, or see below for a discussion and screen snapshot of the Registers Inspector.

Note that the design of the Registers Inspector has changed since the demo video. Most significantly, is no longer possible to set the current thread selection from the Registers Inspector; the tabs at the top of the view have been removed. Thread selection is now done only from the *Threads Inspector*.


Registers: main [3] (Breakpoint)
Memory View
Name Value Region
RAX ffffc8002d669c8 Code-Runtime
RCX 0000000000000008
RDX ffffc8000e84d18 Heap-Boot
RBX ffffc7fdfcb1120 Heap-To
RSP ffffc7ffecfdd18 Thread-2 Stack
RBP ffffc7ffecfdd28 Thread-2 Stack
RSI ffffc7fdfcdb4d8 Heap-To
RDI ffffc8000e84d18 Heap-Boot
R8 0000000000000000
R9 ffffc80017c4f38 Heap-Boot
R10 0000000000000000
R11 ffffc7ffecfcec8 Thread-2 Stack
R12 0000000000000000
R13 0800000defe5c948
R14 00000000004lc118 Thread-2 Loca...
R15 0800000defe5c948
XMM0 5.55366086E-315d
XMM1 5.55366086E-315d
XMM2 5.24282163E-315d
XMM3 1.0d
XMM4 0.0d
XMM5 0.0d
XMM6 0.0d
XMM7 0.0d
XMM8 0.0d
XMM9 0.0d
XMM10 0.0d
XMM11 0.0d
XMM12 0.0d
XMM13 0.0d
XMM14 0.0d
XMM15 5.24282163E-315d
RIP ffffc8002d669d5 Code-Runtime
FLAGS _____I_S____...

The Registers Inspector displays the register contents in the VM for the thread that is currently selected in the Threads Inspector. The name, ID, and state of this thread appear in the title bar of the Registers Inspector window.

The Registers Inspector displays a list of name/value tuples, described by the following columns that appear in the example to the right:

- *Name*: a string identifying the register, derived from an architectural description of the target machine for which the VM was built.
- *Value*: the current contents of the registers, refreshed by reading from the VM process each time the VM halts. Number-valued register values are displayed with numerous visual and interactive behaviors that depend on the value, as described elsewhere (see *Memory Word Values*).
- *Region*: the standard, optional *Memory Region Column*, which identified the known memory region, if any, into which the current register value points.

When a register changes value after a VM execution (either by single step or by running to a breakpoint), attention is drawn to that register in this Inspector by coloring the Name in red. With each successive execution of the VM, as the value ages, the register name migrates from red, though magenta, then blue, and finally black.

A mouse middle-button click over one of the values in XMM registers cycles the Value display through three modes: hexadecimal, single-precision float, and double-precision float; (see *Memory Word Values*). Since the VM only uses the lower 64 bits of XMM registers, the inspector display only this part.

The selection of visible columns can be made using a dialog created by the View Options entry in the Inspectors View menu. Additional columns available include Address and the standard Memory Region Column.

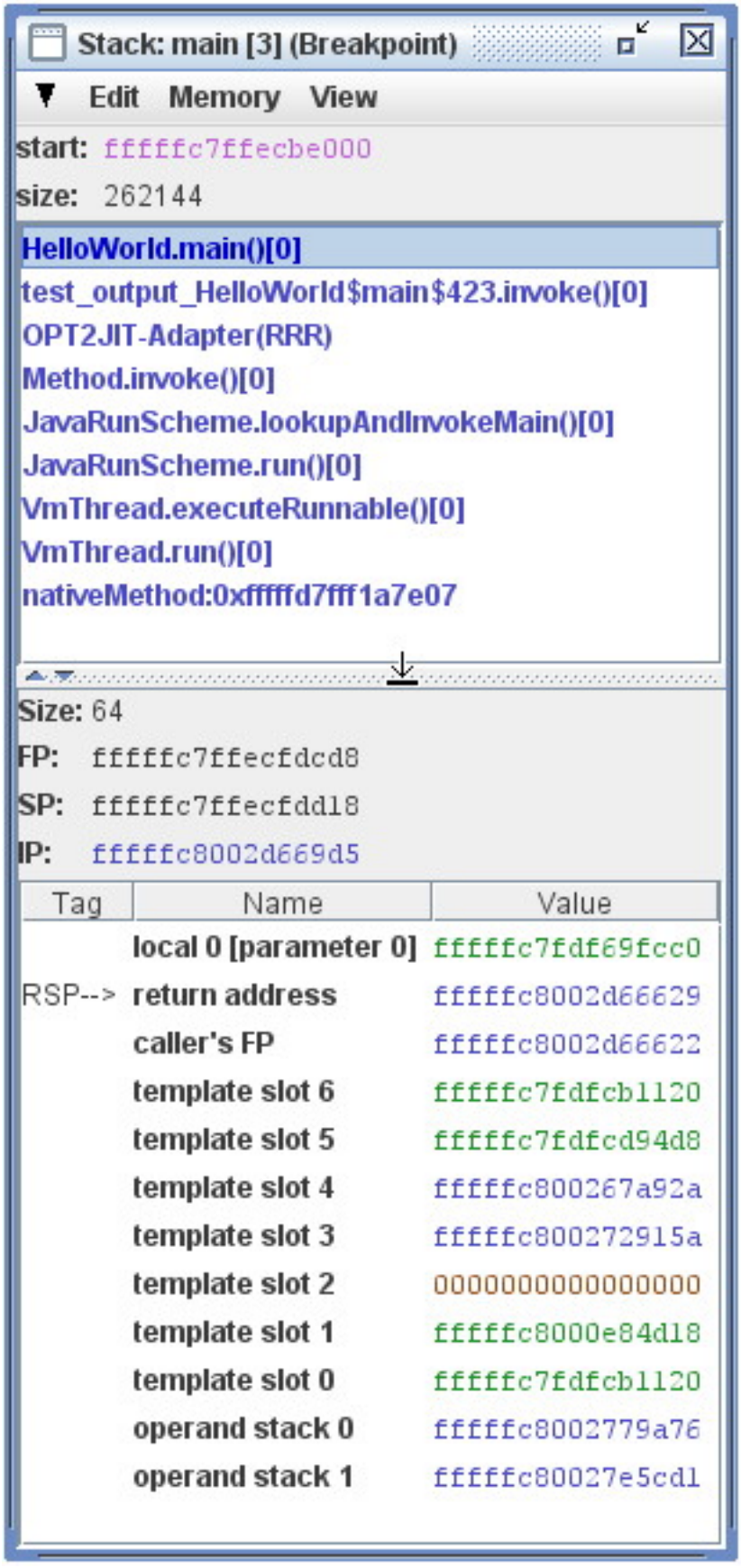
Stacks

The Stack Inspector is one of the Maxine Inspector's tools for examining machine state during execution of the Maxine VM.

View a short demo here, or see below for a discussion and screen snapshot of the Stack Inspector.

Note that the design of the Stack Inspector has changed since the demo video. Most significantly, it is no longer possible to set the current thread selection from the Stack Inspector; the tabs at the top of the view have been removed. Thread selection is now done only from the *Threads Inspector*.

The Stack Inspector displays the stack and currently selected stack frame in the VM for the thread that is currently selected in the Threads Inspector. The name and status of this thread appears in the title bar of the Stack Inspector window.



Each Stack Inspector displays several kinds of information:

- The *Memory location* of the stack is expressed at the top of the display as a start memory address,

expressed in hexadecimal, and `size` in bytes in the VM.

- The middle of the display lists the *stack frames* currently on the stack, identified by unqualified method name and compilation sequence identifier. The currently active method appears at the top; native code about which nothing is known is identified by memory address of the entry. The stack can have a single *selected frame*, `HelloWorld.main()0` in the example, selected with a mouse left-click over the list entry.
- The bottom pane of the display describes the currently selected stack frame, including a list of slots contained in the frame.

Selecting a stack frame causes it to become the *currently selected stack frame* (see *User Focus*). It has the side effect of creating a Method Inspector for the method with a *Machine Code view*, a *Bytecode view*, or both, depending on user preferences and availability of the two representations.

The currently selected stack frame

The contents of the currently selected stack frame appear at the bottom of the view, beginning with specific information concerning the size of the frame and certain key pointers: `FP`, `SP`, and `IP`. Below those appears a list of the stack slots in the frame, described by default using three columns:

- *Tag*: as with Tag columns in other Inspectors, lists any registers in the currently selected thread that point at this location, along with a possibly set *Watchpoint*. A mouse double-left-click in the Tag column sets a watchpoint at the specified location.
- *Name*: a symbolic name of the slot, derived from internal descriptions of the frame layout.
- *Value*: the current value in the memory location, refreshed from VM memory after each execution; displayed with numerous visual and interactive behaviors that depend on the value, as described elsewhere (see *Memory Word Values*).

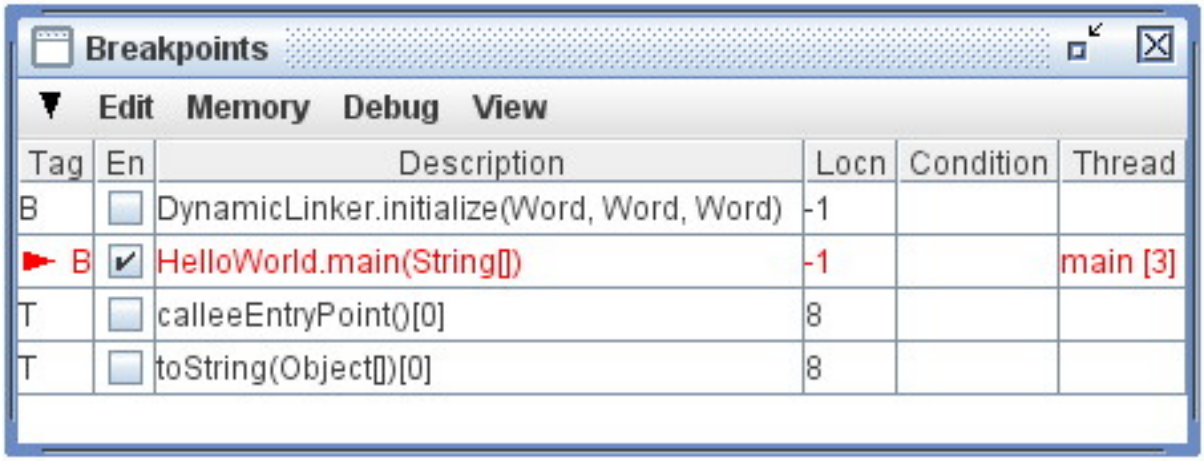
The selection of visible columns can be made using a dialog created by the View Options entry in the Inspectors View menu. Additional columns available include *Address*, *Offset*, and the standard *Memory Region Column*.

Breakpoints

Debugging with the Maxine Inspector is facilitated by a polymorphic approach to code breakpoints that is still very much under development. Just as code can be viewed in more than one way (*Machine Code*, *Bytecode*, and eventually source code - see *Multi-code Method Views*, debugging will likewise be carried out in terms of more than one level of code view.

View a short 2008 demo here, or see below for a discussion and screen snapshot of the Breakpoints Inspector.

Note that the design of the Breakpoints Inspector has changed somewhat since the demo video.



The Breakpoints Inspector lists all breakpoints that exist in the current session and displays their status, by default with the following columns:

- *Tag*: specifies whether the breakpoint is expressed in terms of a machine code (method compilation) location in memory ("T") or as a bytecode location ("B"). The implementation of bytecode breakpoints is incomplete at this time, and source code breakpoints are not yet supported. The column also displays a pointer at the breakpoints, if any, that currently have blocked a thread.
- *En*: a checkbox that can be used to enable/disable a specific breakpoint.
- *Description*: identifies the Java method in which the breakpoint is set, and a mouseover Tooltip provides more detailed information.
- *Locn*: describes the position in the method code at which the breakpoint is set, expressed in bytes from the method entry. A value of -1 denotes abstractly the entry of a method, even if little about the method is known.
- *Condition*: an editable field in which an expression can be supplied that makes the breakpoint conditional, supported at present only for machine code breakpoints
- *Thread*: identifies a thread, if any, that is currently stopped at the breakpoint.

A mouse left-click over a row in the Breakpoints Inspector causes it to become the currently selected breakpoint (see *User Focus*). It has the side effect of making the breakpoint's code location visible in a Method Inspector showing the appropriate kind of code (machine code or bytecode); it also selects the instruction at that location. A special colored box appears in the Tag column of a code view at the location of a breakpoint.

Setting breakpoints

The Maxine Inspector provides a number of ways to set and clear (delete) breakpoints:

- commands on the standard *Debug menu*;
- commands in the Edit menu on the Breakpoints Inspector's menu bar;
- debugging command buttons on Method Inspector code views (for example mouse left-double-click over a code instruction; and
- by keyboard shortcuts.

The semantics of machine code and bytecode breakpoints differ, most notably because there can be 0, 1, or many compilations of a single method. A machine code breakpoint is anchored at a specific memory

location in the code region, and is thus in effect for only that specific compilation, whereas a bytecode breakpoint should in principle be in effect for every compilation of the method, current and future.

Breakpoints persist across sessions as long as the same boot image is being used; that restriction may be eliminated in the future for some kinds of breakpoints.

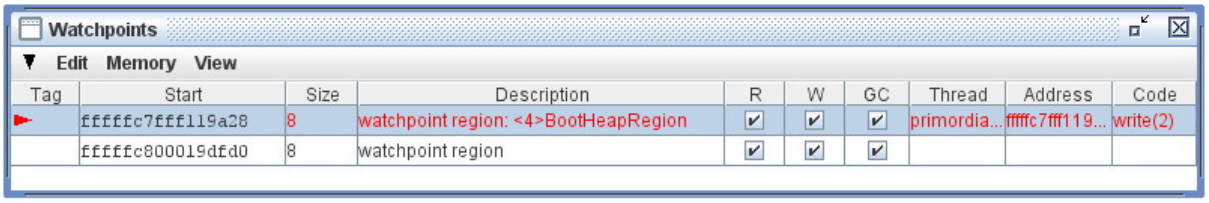
Watchpoints

Debugging with the Maxine Inspector is facilitated by a watchpoint mechanism, supported on some platforms, that is currently under development. It is possible to place watchpoints that catch reads and/or writes and/or executions taking places at specified memory locations.

Because debugging in the presence of relocating Garbage Collection is especially problematic, the Inspector's watchpoint mechanism supports specific features above and beyond conventional watchpoint behavior:

- a watchpoint may be specified either in terms of an *absolute memory location* (the conventional mode) or in terms of a *specific object and its fields*. The latter are known as Maxine object watchpoints, and they will be automatically relocated by the Inspector when the object's representation in VM memory is relocated by GC.
- a watchpoint may be configured to be either active or inactive during execution periods when GC is underway; this can help suppress spurious watchpoint triggers caused as a side effect of ordinary rearrangement of memory by GC.

The Watchpoints Inspector lists all watchpoints that exist in the current session and displays their status, as shown in the following example.



The columns visible in this example include:

- *Tag*: displays a red pointer at the watchpoint, if any, that has blocked a thread.
- *Start*: the VM memory address at which the watchpoint starts.
- *Size*: the amount of memory in bytes covered by the watchpoint.
- *Description*: a string describing how the watchpoint was created:
  - "watchpoint region" if set only at a specified memory location, or
  - containing a description of an object if set on one or more fields of an object; in this case the location of the watchpoint will be updated automatically whenever the representation of the object is moved in VM memory by GC.
- *R*: a checkbox controlling whether the watchpoint should trigger when the memory location is read.
- *W*: a checkbox controlling whether the watchpoint should trigger when the memory location is written.
- *GC*: a checkbox controlling whether the watchpoint should trigger at all during VM execution periods when GC is operating.

- *Thread*: the thread, if any, that triggered the watchpoint.
- *Address*: the specific location, if any, that triggered the watchpoint.
- *Code*: an indication of what action in the VM caused the trigger.

Optional display columns

Additional columns may be displayed via a dialog produced by the *View Options* entry in the Inspector's *View menu*. They include:

- *X*: a checkbox controlling whether the watchpoint should trigger when the memory is read for execution, false by default.
- the standard *Memory Region Column*.

Setting watchpoints

Watchpoints may be created and managed in several ways:

- using entries in the Edit menu on the Inspectors menu bar.
- by mouse left-double-click over the Tag column in any view for which rows correspond to memory ranges.
- using a menu produced by mouse right-click over the Tag column in any view for which rows correspond to memory regions.

Debugging

The Maxine Inspector supports a number of debugging features, most of which leverage the views that have been described in other segments. This section describes how to use those features to handle some specific situations. More features for debugging and more discussion of this topic are forthcoming.

Debugging Traps

To find out when a hardware trap happens, you can set a machine code breakpoint in the responsible trap handler. For example, when you are interested in `SEGV` signals, set a Breakpoint in `com.sun.max.vm.runtime.Trap.handleSegmentationFault()`, using the menu item `Debug` → `Break` at `Method Entry` → `Compiled Method...` or its keyboard shortcut `CTRL-SHIFT-E`. Once the VM is stopped at the breakpoint, remove it and place breakpoints at every exit from the method (RET instruction on x64). Then resume the process.

Once one of the latter breakpoints hits, the VM now has left information about the trap in thread local storage where the inspector can pick it up. Select `View` → `Stack` from the menu. This brings up a *Stack inspector* that shows the stack how it was when the trap happened. You can even find out which instruction was responsible by clicking on the top frame. The instruction will be highlighted by a blue selection border.

Note that the *Registers Inspector* will not show register values from the trap site, but the current register values in the trap handler.

Since some exceptions (e.g. null pointer, divide by zero) are implemented as implicit exceptions that are handled via traps, not every trap is necessarily a VM crash, but it may also be normal operation.

To observe divide-by-zero traps, use a different trap handler, then apply the same procedure as above.

Debugging through a Garbage Collection

If you set a breakpoint inside the GC implementation and suspend the VM there, you may find that some object references are no longer functioning. The Inspector detects when a GC is underway and then distinguishes references into the boot image from those into the runtime heap. Whereas the former are immutable and always intact, the latter are considered broken during GC. Where ever they appear in fields, array elements etc. they change in color from green to red and most interaction with them is disabled.

Once the GC has finished, the Inspector refreshes all the red references and they become green and fully functional again.

Native Code

The Inspector provides some limited support for debugging native code that is either included in the VM image or loaded dynamically by user code, e.g., with `System.loadLibrary`.

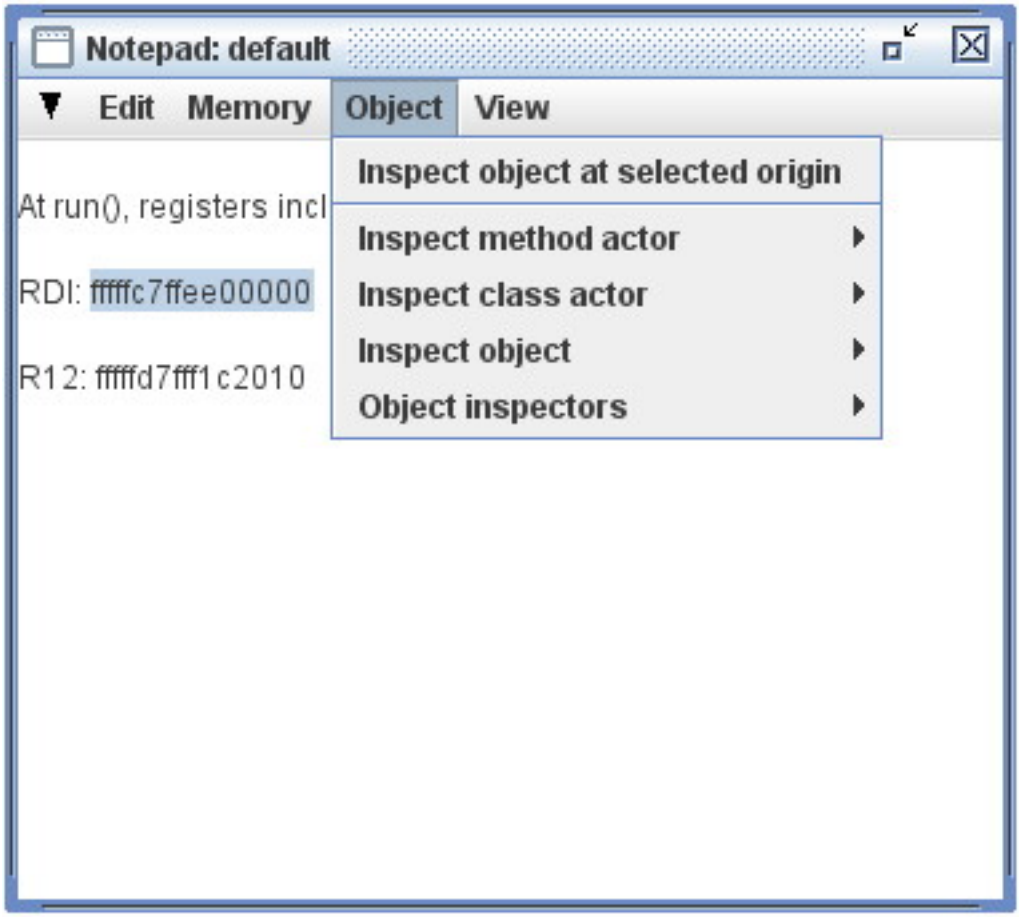
The Code menu provides an entry View native function by name. This initially brings up a dialog with a list of loaded libraries, for example, `libjvm` which contains the native code that supports the VM. Selecting a library then brings up a list of functions defined in the library. Selecting a function then brings up a code view for that function. These dialogs behave similarly to those for Java methods so that name filtering, for example, works as expected. Note that if this menu entry is invoked before a symbol lookup in the library has occurred, the addresses of the functions in a library will not be known. In this case a dialog with the text "Functions are not available at this stage" is displayed. This window is usually very short and is unlikely to be encountered in practice.

A breakpoint may be set at the entry to a native function by the `Debug -> Break` at `machine code -> Native function`, which brings up similar dialogs.

There is currently no support for symbolic display of instructions within a native function, and stack walking is not implemented within a chain of nested function calls.

Notepad

The Inspector provides a persistent notepad for the user's convenience. The notepad contains an arbitrary collection of text that the user can manage and which endures across restarts of the Inspector. A small amount of specialized behavior driven by the contents of the notepad is supported, and this may be extended in the future.



Viewing and editing access to the Inspector's notepad is provided by the Notepad Inspector, an example of which appears to the right. This Inspector is a very simple text editor whose single buffer is implicitly persistent; it need not be explicitly saved, and it will always endure across Inspector sessions.

Text in the Editing commands include the familiar *Cut*/*Copy*/*Paste* commands, which interoperate with the same system clipboard used by the rest of the Inspector and which are available in three ways:

- from entries on the Inspector's Edit menu;
- from entries on a menu that pops up in response to a mouse right-click over the editing area; and
- from conventional keystroke accelerators, which are noted in the menu entries.

Specialized behavior

Additional specialized behavior is available when a selected range of text can be interpreted as a memory address expressed in hexadecimal, as is the selection in the example:

- the commands *Inspect memory at selected address* and *Inspect memory region containing selected address* become enabled and can create *Memory Inspectors* as suggested by their names. These commands appear on a menu that pops up in response to a mouse right-click over the editing area and as context-specific additions to the standard *Memory* menu on the menu bar. When the selection cannot be interpreted as a memory address, the commands are disabled and grayed out.
- when the selected range of text can be interpreted as a memory address that is the origin of a VM *Heap Object*, the command *Inspect object at selected origin* becomes enabled and can create an *Object Inspector* displaying the object's representation. This command appears on a menu

that pops up in response to a mouse right-click over the editing area and as a context-specific addition to the standard *Object menu* on the menu bar (see example). When the selection cannot be interpreted as a memory address that points at an object origin, the command is disabled and grayed out.

User Focus

The views (known as individual "Inspectors") available within the Maxine Inspector make visible many different aspects of the VM state (see, for example, the Inspectors listed on the standard View menu), and many of them support some kind of user-driven selection. Some selections have side effects that cause other Inspector views make visible information related to the selection; in other words, some view actions are coordinated by user selections. The mechanism for this coordination is the *user focus*. When the description of an Inspector refers to the *currently selected X*, for some kind of VM entity X, it refers to the specific instance of X that is set in the user focus.

Some user selections have the side effect of setting a *user focus*: a selection that is shared among all Inspector views. At present, such shared selections include:

- *Thread*
- *Stack Frame*
- Code Location (*Machine Code*, *Bytecode*, or both)
- *Breakpoint*
- *Watchpoint*
- *Memory address*
- *Heap Object*

Strict View Coordination

In the case of threads, the coordination among views is strict, so that there is an invariant relationship among certain thread-specific views:

- The *Registers Inspector* only displays the registers for the *currently selected thread*.
- The *Stack Inspector* only displays the stack and its stack frames for the currently selected thread.
- The *Thread Locals Inspector* only displays thread local storage for the *currently selected thread*.
- *Machine Code Inspectors* and *Bytecode Inspectors* display the instruction pointer and call return sites on the stack only for the *currently selected thread*.
- Any *Tag* column in a memory-related view adds annotations only for registers in the *currently selected thread* that point into the row's memory region.

Another example of strict coordination occurs within the *Stack Inspector*. The lower part of the Inspector only displays the stack frame slots for the currently selected stack frame.

Relaxed View Coordination

Some selections that set the user focus have side effects on other views, but there is no strong invariant of the sort mentioned above for threads. The goal of these side effects is to bring into user view some

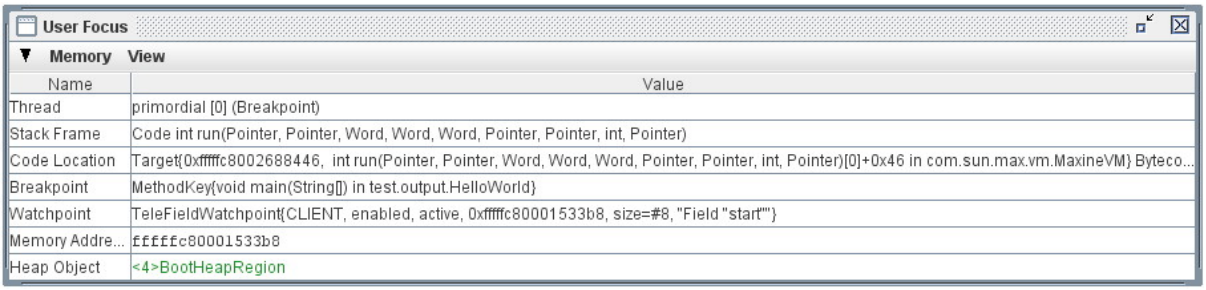
relevant information about the entity just selected. The specific nature of these rules is evolving, based on user experience. Examples include:

- Selecting a frame in the *Stack Inspector* causes the *instruction pointer* in the frame (if the top frame) or *call return pointer* (if other than the top frame) to become the currently selected code location. This in turn causes the appropriate Method Inspector to display the method and select the code location in that view.
- Selecting a breakpoint causes its location to become the currently selected code location, which in turn causes the appropriate Method Inspector to display the method and select the code location in that view.

Subsequent actions, for example selecting another code instruction, will break these relationships.

The User Focus Inspector

A specialized inspector displays the current members of the *user focus* at any time. In the example below, every aspect of the user focus is non-null, but this is not always the case.



This Inspector is intended mainly for testing the Maxine Inspector itself (which is why it appears only in the Test menu on the main menu bar), but it can be useful to help understand unexpected interactions.

Menus

The Maxine Inspector displays menus in three contexts:

- the *main menu bar*, which appears at the top of the entire application frame;
- an *inspector menu bar* on each of the specific Inspector views; and
- a *popup menu* that appears in response to a mouse right-click.

A distinguished set of *standard menus* can appear in many contexts and have behavior that is generally independent of context. In other words these menus have the same names, entries, and behavior no matter where they appear. The standard menus, described in more detail below, are named Memory, Object, Code, Debug, and View.

Other menus have behavior that is generally dependent on context, for example the Default menu that is accessible at the upper left of every Inspector view under a triangle icon, and the Edit menu that appears on Inspectors where the displayed contents can be modified in some way.

Finally, some menus are a combination of a standard menu with additional context-dependent entries added before the standard entries, separated by a horizontal separator.

The Main Menu Bar

The standard menus Memory, Object, Code, Debug, and View appear, among others, in the main menu bar of the Inspector, as shown below:



Other menus, unique to the main menu bar include:

- *Inspector*: general functionality, including Refresh all Views, Close all views, Preferences, and Quit Inspector.
- *Java*: very specialized commands concerning the VM, for example setting tracing level in the VM.
- *Test*: some commands specialized for debugging the Inspector. Most list some summary of internal state to the console. The exception, View User Focus, creates the User Focus Inspector, which summarizes all the aspects of current user focus, for example selected selected thread, stack frame, selected memory address, etc. See User Focus.
- Helep: access to the Inspector's Help System would be here, if it had one. Sorry.

Inspector Menu Bars

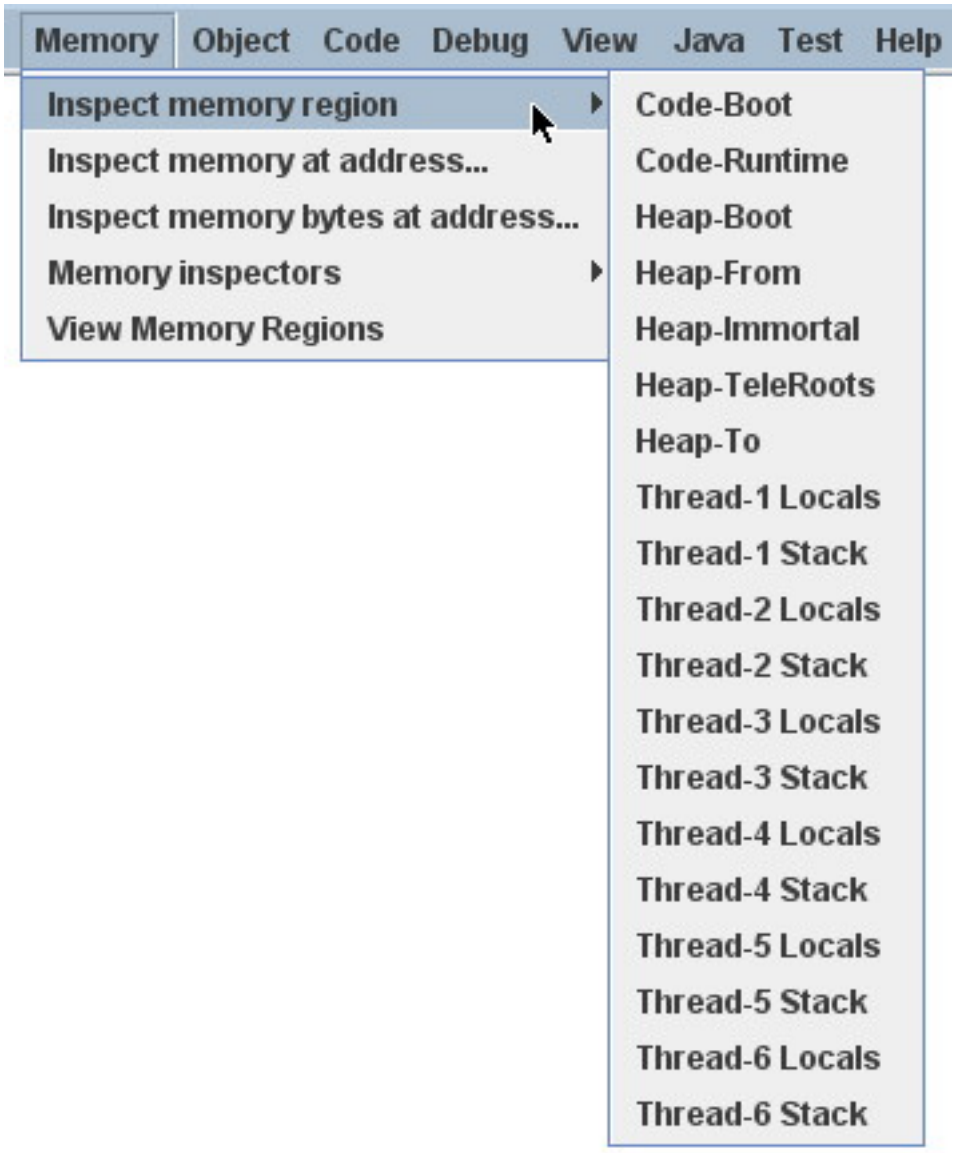
Standard menus sometimes appear (when relevant) in the menu bar of individual Inspector windows. In such cases, the menus sometimes contains additional menu entries that are dependent on the context of the particular view. These context-dependent menu entries usually appear first on the menu, followed by a line that acts as a separator, followed the the standard context-independent entries. For example, in the example display of the *Notepad Inspector*, the standard Object menu contains an additional entry that is sensitive to the current text selection in the notepad.

The Standard Memory Menu

The standard Memory menu contains entries designed to create and manage Inspectors related to low-level memory properties of the VM. They include:

- *Inspect memory region*: produces a dynamically generated submenu listing all known *Memory Regions*. Selecting an entry produces a *Memory Inspector* whose display spans the region.
- *Inspect memory at address...*: produces a dialog in which a memory address can be entered, which in turn produces a *Memory Inspector* whose display begins at the specified address.
- *Inspect memory bytes at address...*: produces a dialog in which a memory address can be entered, which in turn produces a *Memory Bytes Inspector* whose display begins at the specified address.
- *Memory inspectors*: produces a dynamically generated submenu listing all existing *Memory Inspectors*. Selecting an entry brings the Inspector window to the foreground.

- *View Memory Regions*: produces the *Memory Regions Inspector*. This entry also appears on the standard View menu.



As with all standard menus, the standard Memory menu also appears on some individual Inspector windows. When it does appear, it often contains some additional entries that are context-dependent, which is to say their behavior depends on the particular Inspector in which it appears. For example, each *Object Inspector* adds the entry *Inspect this object's memory*, which produces a *Memory Inspector* whose display spans the representation of the object being inspected.

The Standard Object Menu

The standard Object menu contains entries designed to create and manage Object Inspectors. They include:

- *Inspect method actor*: produces a dialog that permits identification (by name) of a Java method presumed to be loaded into the VM. If so, this produces an *Object Inspector* on the special VM object (of type MethodActor) that the VM uses to represent information about the method.
- e: produces a dialog that permits identification (by name or ID) of a Java class presumed to be

loaded into the VM. If so, this produces an *Object Inspector* on the special VM object (of type ClassActor) that the VM uses to represent information about the class.

- *Inspect object*: produces a dialog that permits identification of a Java object (by address or Inspector session ID) presumed to exist into the VM. If so, this produces an *Object Inspector* on the object.
- *Object Inspectors*: produces a dynamically generated submenu listing all existing *Object Inspectors*. Selecting an entry brings the Inspector window to the foreground.

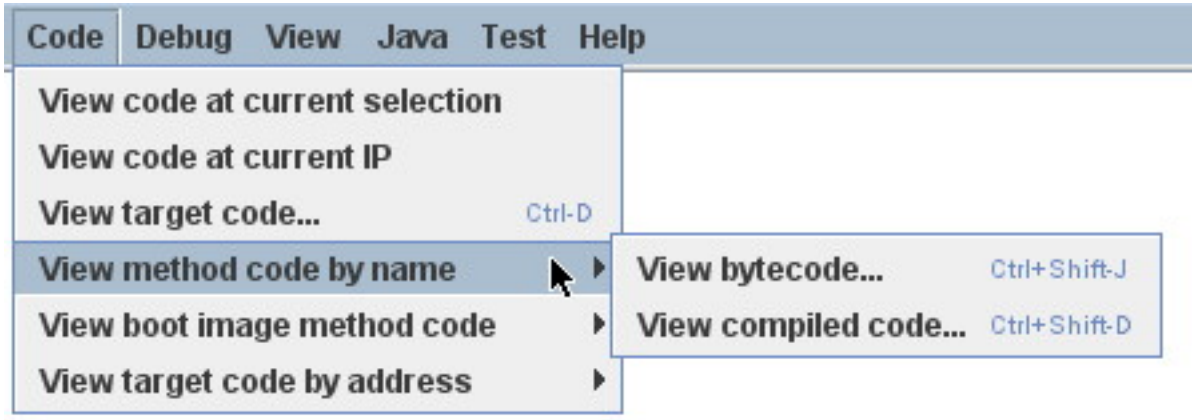


As with all standard menus, the standard Object menu also appears on some individual Inspector windows. When it does appear, it often contains some additional entries that are context-dependent, which is to say their behavior depends on the particular Inspector in which it appears. For example, each Method Inspector adds entries that create *Object Inspectors* for VM objects that represent important information about the method being viewed: class, method, compilation, etc.

The Standard Code Menu

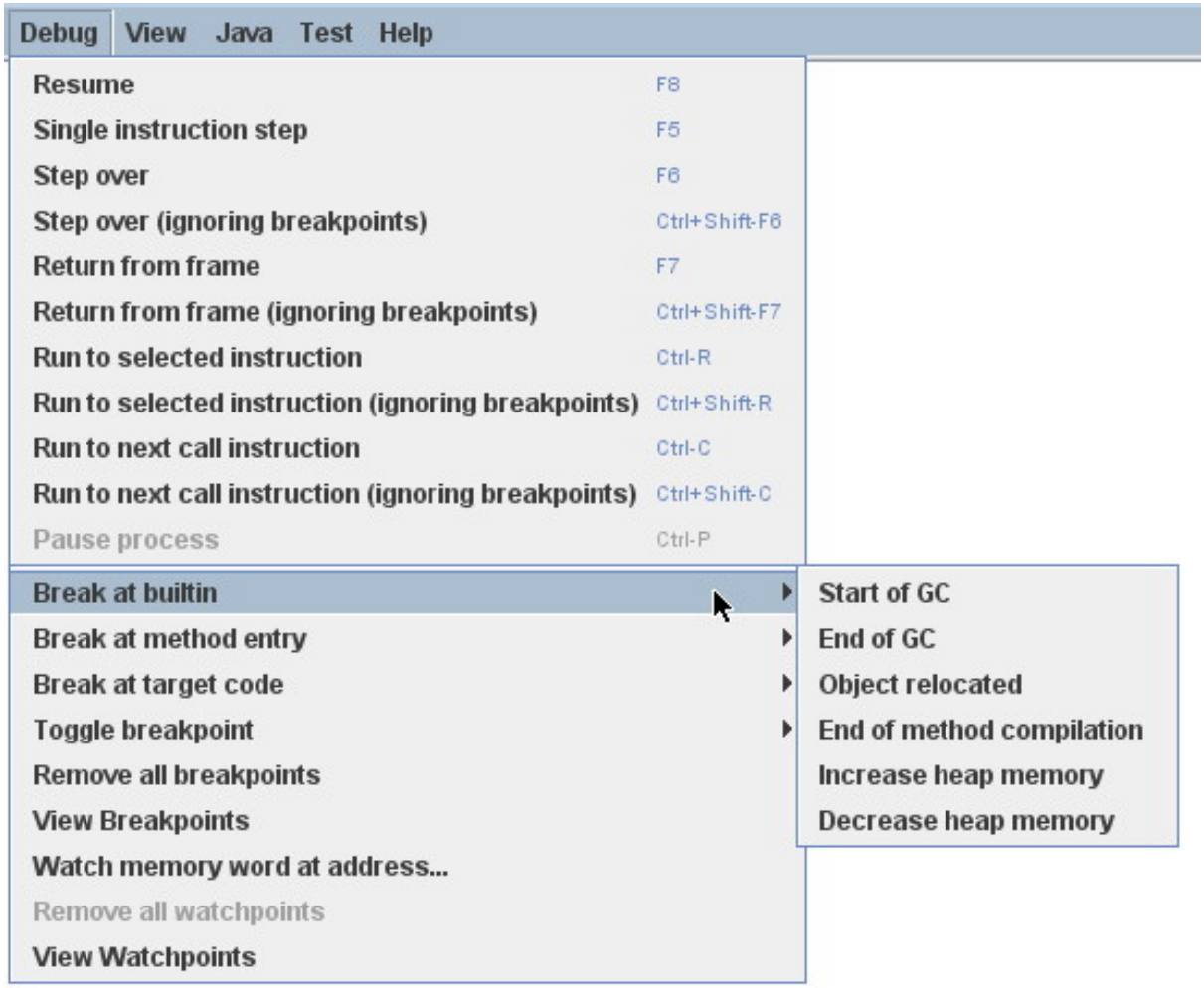
The standard Code menu offers a number of ways to locate and view code in the VM:

- *at current selection*: the currently selected code location (see *User Focus*), if set.
- *at current IP*: at the code location of the instruction pointer in the currently selected thread.
- *target code...*: a method selected interactively from all known compiled methods, first by class and then by method. Typing into a filter field in these dialogs makes them a very fast way to find an existing compilation.
- *method code by name*: a method described interactively by specifying a name.
- *boot image method codew*:
- *target code address*: compiled code located at a memory address entered interactively into a dialog.



The Standard Debug Menu

The standard Debug menu is unusual in that contains two categories of entries (both related to debugging), separated by a horizontal line.



The first category of menu entries provide debugging control of the VM process: *Resume*, *Step* (sometimes known as "step in"), *Step over*, *Return* (sometimes known as "step out"), *Run to selected...*, *Run to next call...*, *Pause*, and variations.

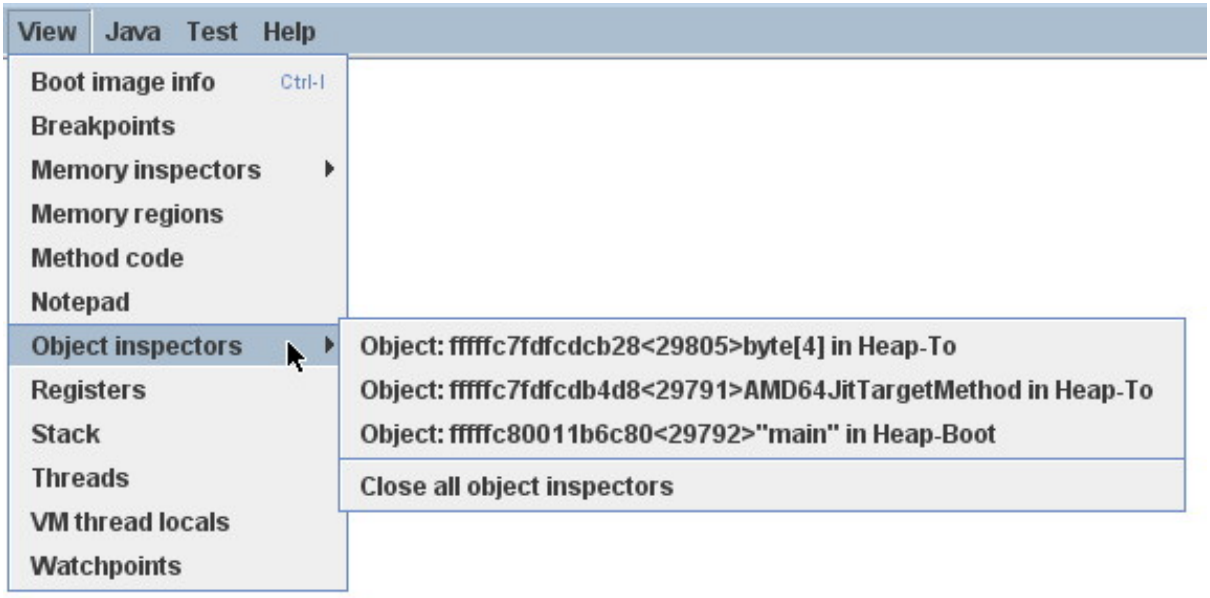
The second category offers management of *breakpoints* and *watchpoints* in the VM's process.

- *Break at builtin*: produces a submenu containing predefined locations in the VM at which breakpoints can be set; this is a convenience for setting breakpoints at significant VM events, without requiring that the user know in advance exactly what method represents the action.
- *Break at method entry*: produces one of a variety of dialogs for specifying methods at which an entry breakpoint should be set.
- *Break at target code*: produces one of a variety of dialogs for specifying locations in memory at which a machine code breakpoint should be set.
- *Toggle breakpoint*: turns on or off the breakpoint at the currently selected code location (see *User Focus*).
- *Remove all breakpoints*: (enabled only when there are breakpoints set) clears all breakpoints from the VM process.
- *View Breakpoints*: creates the *Breakpoints Inspector*.
- *Watch memory word at address*: produces a dialog in which the use can enter a specific memory address in hexadecimal, which will be used as the origin of a newly created *Object Inspector*.
- *Remove all watchpoints*: (enabled only when there are watchpoints set) clears all watchpoints from the VM process.
- *View Watchpoints*: creates the *Watchpoints Inspector*.

The Standard View Menu

The standard View menu provides access to all the different kinds of Inspectors that are available during a Maxine inspection session. Most are singletons, in which case the specified Inspector is either created or simply brought to the front if it already exists. In the two cases where there can be any number of inspectors (Memory and Objects), submenus are dynamically generated that allow a specific inspector to be brought to the front.

- *Boot image info*: produces the *Boot Image Inspector*
- *Breakpoints*: produces the *Breakpoints Inspector*
- *Memory inspectors*: produces a dynamically generated submenu listing all existing *Memory Inspectors*. Selecting an entry brings the Inspector window to the foreground.
- *Memory regions*: produces the *Memory Regions Inspector*
- *Method code*: produces the *Method Inspector*.
- *Notepad*: produces the *Notepad Inspector*
- *Object Inspectors*: produces a dynamically generated submenu listing all existing *Object Inspectors*. Selecting an entry brings the Inspector window to the foreground.
- *Registers*: produces the *Registers Inspector*
- *Stack*: produces the *Stacks Inspector*
- *Threads*: produces the *Threads Inspector*
- *VM thread locals*: produces the *VM Thread Locals Inspector*
- *Watchpoints*: produces the *Watchpoints Inspector*



As with all standard menus, the standard View menu also appears on individual Inspector windows. When it does appear, it contains some additional entries that are context-dependent, which is to say their behavior depends on the particular Inspector in which it appears. The two that are present in most cases are:

- *View options*: produces a dialog that permits setting persistent *User Preferences* related to the views. In cases where multiple Inspectors of a given kind can exist, the dialog permits setting each preference either temporarily (just for the instance being viewed) or persistently for all subsequently created Inspectors. These include inspectors for Memory, Methods, and Objects.
- *Refresh*: causes data being displayed in this specific view to be reread from the VM and redisplayed.

The Default ("Triangle") Inspector Menu

Every Inspector Window contains a "Default" menu with generic command that relate mostly to windows, for example *Close*, *Close Other Inspectors*, *Refresh*, and sometimes *Print*.



Some default menus contain additional entries with more specific, context-dependent behavior. For example, the default menu in the Object Inspector example to the right contains the two commands

Close other object inspectors and *Close all object inspectors*, and the default menu in every Memory Inspector contains analogous entries.

The Edit Menu

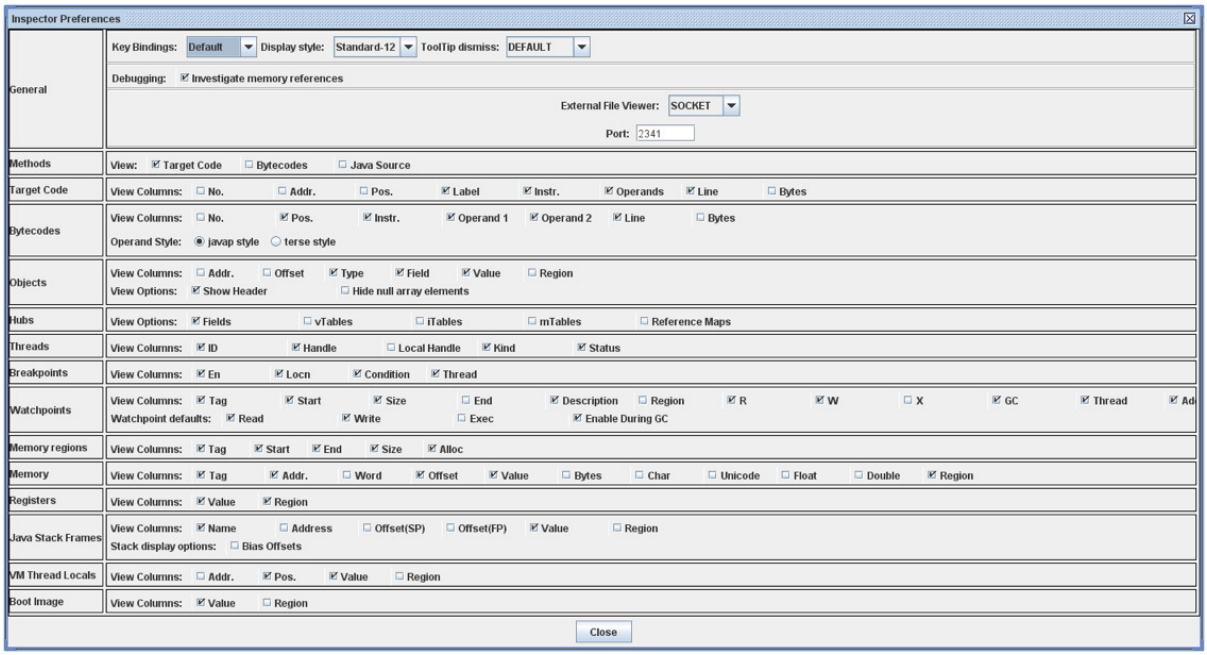
Every Inspector views in which the contents can be changed will display an appropriate Edit menu in the menu bar. For example the Edit menu in a Memory Inspector contains a command to change the origin, and the Edit menu in both the Breakpoints Inspector and Watchpoints Inspector have commands to delete breakpoints and watchpoints respectively.

User Preferences

Most of the Inspector's views provide user selectable view options that configure the information appearing in the displays. In some cases these preferences can be set for either a specific instance (for example a particular *Object Inspector*, or as a general preference for all subsequently created views of that kind.

The view options relevant to each Inspector are by convention available via a menu item named View Options on the window frame of the particular Inspector. A summary of all view options, as well as other user preferences, can be managed invoking the Preferences action on the Inspector's main menu bar. The image below displays the current appearances of the dialog for managing user preferences.

The settings of these preferences are made persistent by default, stored in a file typically named `maxine.ins`.



7.24.5 How the Inspector interacts with the Maxine VM

This page describes how the *Maxine Inspector*'s interaction with a running VM is *implemented*.

General goals for the Inspector all deal with making development and experimentation in the Maxine VM as productive and widely accessible as possible:

- Support basic debugging of the Maxine VM, something not possible with standard tools.

- Make visible as much internal VM state as possible, both in terms of design abstractions and in terms of low-level representations, even when the VM is broken.
- Provide new developers with a fast path to understanding VM internals.

A few general strategies guide the Inspector's implementation:

- Run in a separate process (usually local, but see Guest VM), so that inspection depends neither on a correctly running VM process nor there being any process that all.
- Require as little active support from the VM as possible, in particular require no active agent.
- Reuse as much VM code as possible, especially reading/writing/understanding the low-level representations of data on the target platform (possibly different than the platform on which the Inspector runs).
- Load VM classes into the Inspector for reflective use in understanding VM data.
- Rely on platform-specific implementations for low-level interaction with a running VM: process control, threads, breakpoints, access to memory and registers.

This section describes the Inspector's access to the lowest level abstractions in the VM process, namely the resources provided by the underlying platform: memory, threads, and execution state.

One of the most difficult and frustrating parts of the Inspector's implementation is the need to implement low-level process controls on the several supported platforms. These controls include reading and writing from the VM's memory, finding and inspecting threads, setting breakpoints, setting watchpoints, and deciphering process state.

Generic controls are implemented in class `com.sun.max.tele.debug.TeleProcess`. Concrete subclasses using native methods implement the controls for specific platforms:

- *Solaris*: platform support is best on Solaris, where libproc provides a programmatic interface to the Solaris `/proc` pseudo-filesystem. Watchpoints are supported with no limit on their number (see `Native/tele/darwin/*.[ch]`).
- *Linux*: the Inspector uses a mixture of `ptrace(2)` and `/proc` (see `Native/tele/linux/*.[ch]`).
- *Mac OS X*: on the Mac the Inspector uses a mixture of `ptrace(2)` and the Mach API (see `Native/tele/darwin/*.[ch]`).
- *Guest VM*: the Guest VM variant of the VM runs in a Xen domain where such OS services are unavailable, so controls must be implemented using Xen inter-domain communication.

This code can be very subtle. It now seems to work fairly reliably, but at the cost of many hours deciphering non-documentation and gdb source code. In our experience, programming a debugger is a very niche activity.

Low-level memory access is implemented using basic process control methods in class `TeleProcess`:

```
read(Address address, BtyeBuffer buffer, int offset, int length)
write(Address address, ByteBuffer buffer, int offset, int length)
```

However, interpreting the bits presents more of a challenge, since this must be done for a VM running on a potentially different platform. Fortunately, the Inspector is able to load the Java classes that describe the target platform and then reuse the VM's own code for reading and writing bits representing the VM's internal primitive data types. Methods for reading and writing those types appears in interface `com.sun.max.tele.data.DataAccess`, and all but the lowest-level read methods are implemented by class `com.sun.max.tele.data.DataAccessAdapter`.

For performance reasons, especially for non-local debugging such as with the Guest VM, the Inspector caches pages of memory read since the most recent process execution (see class `com.sun.max.tele.page.PageDataAccess`).

The Inspector's low-level interaction with the VM process can be observed. See *Low-level logging* for instructions on enabling all low-level VM logging. In order to observe only Inspector-related events, change `log_TELE` to 1 in `Native/share/log.h`, rather than `log-ALL`.

Although the Inspector is designed to rely as little as possible on the internals of the VM, there are a number of ways in which the VM is constructed to make inspection as easy as possible. The mechanisms described in this section incur zero runtime overhead in the VM, and involve no writing into VM memory.

The Inspector leverages considerable knowledge of the VM's internal data representations to build its model of VM state, but it must have somewhere to start when beginning to read from a memory image. The boot image generator stores in the boot image header a number of addresses and other data that help the Inspector (and VM) find things. These addresses get relocated, along with the contents of the heap, during *Bootstrap*. The Inspector leverages detailed knowledge of the header's contents in order to locate, among others:

- the VM's *schemes* bindings, which are loaded into the Inspector
- the *boot heap*
- the boot code region
- the class registry
- the list of dynamically allocated heap segments
- the list of thread local areas
- the entry location of key methods

The Inspector uses a variety of mechanisms to locate instance or class fields in the heap. During the Inspector's starting sequence (when little is yet known about VM state), fields are typically located by relying on specific knowledge of a few key object types, possibly using Java reflection on the VM classes (which are all loaded into the Inspector). This kind of access is relatively unsafe, since it bypasses the type system in the running VM. There are more abstract ways to access fields, but they rely on the Inspector's model of VM's class registry, which must first be created using the low-level mechanisms.

The simplest way to exploit higher-level field access mechanisms is to annotate (in VM code) fields of interest using `@INSPECTED`. The main method in `com.sun.max.tele.field.TeleFields` reads VM sources, generates field access methods, and writes them back into itself for use by the Inspector. These access method implementations hide all the indirections necessary to read or write field data (taking into account the hardware platform, the layout being used, the particular representation for the object, and the class layout) and return values of the desired types.

The Inspector uses a variety of mechanisms to locate methods and their compilations (either instance or class). Specific methods can be called out for enhanced access by the Inspector by annotating (in VM code) those methods using `@INSPECTED`. The offline program TeleMethods reads VM sources, generates method access methods, and writes them into class `com.sun.max.tele.method.TeleMethods`. These access method implementations hide all the indirection necessary to locate the annotated methods and their meta-information.

VM methods annotated with `@INSPECTED` can be interpreted by the Inspector (for example, see `TeleMethodAccess.interpret()`). Interpretation takes place in the Inspector's process, but in the execution context of the VM: object references are boxed locations in VM memory, reading/writing

is redirected through VM data access, class ID lookup is redirected to the Inspector's model of the VM's class registry, and bytecodes are located using reflection on the VM's code loaded in the Inspector.

The Inspector's interpreter runs very slowly. It is used routinely by the Inspector in only a few situations, where VM data structures to be navigated are too complex (e.g. a hash table) to be navigated robustly using low-level techniques. For example, see the Inspector method `TeleCodeCache.findCompiledCode(Address)`, which interprets remotely the VM method `Code.codePointerToTargetMethod(Address)`.

Although the interpreter is in principle capable of writing into VM memory, it is not used in any situations where this happens.

Active VM support for inspection is kept to an absolute minimum, but in most cases either incur very little VM overhead or are enabled only when the VM is being inspected. There are several flavors of support mechanisms:

- Distinguished fields, usually static, where the VM records information exclusively for the consumption by the inspector.
- Distinguished methods, usually static and usually empty, called by VM code exclusively as potential breakpoint locations for the inspector; this is a weak kind of event mechanism.
- Special VM memory locations into which the Inspector writes for consumption by specific VM mechanisms.

As a matter of organization, this kind of support is implemented mainly by VM classes in the package `com.sun.max.vm.tele`, but it often imposes some obligations on specific *scheme* implementations, for example to store a value or call a method. These obligations are increasingly specified and documented in scheme definitions.

The remainder of this section describes a few areas of active VM support for inspection.

Many support mechanisms in the VM operate conditionally, depending on the value of static method `com.sun.max.vm.tele.Inspectable.isVmInspected()`. This predicate checks one of the bits in the static field `Inspectable.flags` in VM memory, which can be set in one of two ways:

- When the VM is started by the Inspector, the Inspector sets that bit in VM memory early in its startup sequence (see Inspector method `TeleVM.modifyInspectableFlags()`).
- When the VM is not started by the Inspector, but when it is anticipated that the Inspector might subsequently attach the VM process, a command line option to the VM makes it inspectable.

At present, the VM cannot be made inspectable unless this bit is set early during the VM startup sequence.

The Inspector tracks every class loaded in the VM, as represented by the current contents of the VM's `ClassRegistry`; the Inspector maintains that information using the Inspector class `TeleClassRegistry`.

The Inspector initializes its `TeleClassRegistry` at VM startup, effectively identifying the classes already loaded in the boot heap; it does this by directly reading (using low-level operations that rely on significant knowledge of the data structures involved) the contents of the VM's `ClassRegistry` in the boot heap. As noted earlier, this data structure cannot be read using the more abstract, relatively more type-safe techniques in the inspector because those techniques rely on type information stored in the `TeleClassRegistry`. This is one of many circularities in the Inspector that reflect the underlying meta-circularity of the Maxine VM.

As the VM loads additional classes dynamically, and when inspection is enabled, the VM records them using the following static fields in VM memory:

```
package com.sun.max.vm.tele;

public final class InspectableClassInfo {
    ...
    @INSPECTED
    private static ClassActor[] classActors;

    @INSPECTED
    private static int classActorCount = 0;
    ...
}
```

The Inspector refreshes the `TeleClassRegistry` each time the VM process halts: it checks the VM's count against its cache and reads information from VM memory about any newly loaded classes.

No provision is made for tracking classes that the VM *unloads*. In fact, the VM implements class unloading by garbage collection, and a regrettable consequence of this inspection mechanism is that it prevents class unloading. This is by far the most egregious interference visited upon the VM by the Inspector, and it might be corrected in the future.

Implementations of the Maxine VM's *heap scheme* are obliged to make certain calls, as documented and supported by the scheme's static inner class `com.sun.max.vm.heap.HeapScheme.Inspect`. All of these calls delegate to the VM class `com.sun.max.vm.tele.InspectableHeapInfo`, which provides several kinds of services when the VM is being inspected (described below): heap allocations, object relocations, and events.

Allocated heap segments

An inspectable, static field in the VM class `com.sun.max.vm.tele.InspectableHeapInfo` holds the list of memory regions currently allocated as heap segments. This list is read from VM memory by the Inspector each time the VM process halts; any additional heap segment allocations to the information are tracked in the inspector class `TeleHeap`. This enables the inspector to make a quick first check about whether a VM memory location could hold a valid heap object, and permits a visualization of all memory allocations made by the VM.

Object locations

The Inspector tracks heap objects of interest: sometimes because the user is viewing them, but much more frequently because they represent vital information about the execution state of the VM. In the presence of relocating garbage collection that can take place at any time (with respect to the Inspector), there is no practical way for the Inspector to track object locations without some support from the VM.

When the VM is being inspected, it actively supports object tracking by allocating in VM memory an additional root table: an array of addresses that are treated by garbage collection implementations as roots to be updated as needed when objects move. Entries in this table are treated by the VM as weak references: both to minimize disruption of VM operation and for the Inspector to discover when objects have become garbage. Access to the root table is provided via inspectable static fields in the VM class `com.sun.max.vm.tele.InspectableHeapInfo`.

The Inspector checks the root table each time the VM halts. It does so by reading two static fields in `com.sun.max.vm.tele.InspectableHeapInfo` that are incremented by the garbage collectors: one counts the number of collections initiated so far and one counts the number of collections completed. The Inspector compares those two counters with their previous values. If a new collection

has concluded since the last refresh, then the entire contents of the VM's root table are copied into the Inspector's cache, where they are available for the Inspector's implementation of remote object references. When the Inspector creates a new object Reference, based on a specific address in the VM's heap, that value is added to an empty slot in the Inspector's root table cache and is written through to the corresponding location in the VM's root table.

The Inspector can also observe object relocation directly, if needed, by setting a breakpoint on the following method:

```
InspectableHeapInfo.inspectableObjectRelocated(Address oldCellLocation,␣
↪Address newCellLocation){}
```

This empty method is called each time an object is relocated and it exists for just this purpose.

Heap events

The VM makes it convenient for the Inspector to halt the VM process at certain interesting events. It does so by creating special methods that are called at those times, methods that do nothing in the VM, but which are convenient for the Inspector to set breakpoints. The VM class `com.sun.max.vm.tele.InspectableHeapInfo` contains the following methods of this sort:

- `inspectableGCStarted()`
- `inspectableGCCompleted()`
- `inspectableObjectRelocated()`
- `inspectableIncreaseMemoryRequested()`
- `inspectableDecreaseMemoryRequested()`

The Inspector's breakpoint mechanism requires active support from the Maxine VM's *compilation scheme*. As a machine-level debugger, the natural kind of breakpoint supported by the Inspector (and by the underlying platform) is specified in terms of a memory location in compiled machine code. However, the Inspector also supports breakpoints specified in terms of a method's signature, so-called *bytecode breakpoints*. The Maxine VM runs only compiled code, so a bytecode breakpoint is understood to mean that there should be a corresponding machine code breakpoint set in every compilation of the method, present or future. A bytecode breakpoint can even be set (at location 0) for methods not yet loaded into the VM.

An early implementation of bytecode breakpoints divided responsibility for setting these breakpoints: the Inspector set them for existing compilations and a request was written into a queue in the VM for the runtime compiler, which would create the machine code breakpoints in any subsequent compilation. This approach had an irreconcilable race and was replaced by the simpler approach of halting the VM immediately after every method compilation. The Inspector would compare the compiled method against its current list and set a machine code breakpoint if needed. This implementation proved to incur too much overhead for non-local debugging, notably for Guest VM.

The current implementation (see Inspector class `TeleBytecodeBreakpoint`) halts the VM after method compilations, but filters those events. Each time the Inspector's list of bytecode breakpoints changes, the Inspector writes into VM memory an easily parsed list of textual type descriptors for those classes for which one or more bytecode breakpoints are currently set. Implementations of the VM's *compilation scheme* are required to call a static notification method in the scheme's static inner class `com.sun.max.vm.heap.HeapScheme.Inspect` at the beginning and end of each method compilation. This delegates to VM class `com.sun.max.vm.tele.InspectableCodeInfo`, where the current list of classes is consulted. If the class of the method just compiled is in the list, it results

in a call to the empty method `inspectableCompilationEvent()` where the Inspector can set a breakpoint. Filtering only by class, not by method, results in some false positives, but the mechanism is simple, fully synchronous, and reduces the interruptions more than enough.

Inspector evolution

The Inspector's life began long before the Maxine VM could run usefully, a period during which the novel meta-circular, highly modular architecture was refined and techniques for generating the Maxine *boot image* developed. The Inspector's original role was static visualization and exploration of the binary boot image in terms of the higher level abstractions of the VM, something that could not be done by any existing tool.

As the VM became increasingly able to run through its startup (*bootstrap sequence*), basic debugging features were added: process controls and breakpoints, along with register and stack visualization. The Inspector remained monolithic (with no model/view separation) and single-threaded (the GUI froze during VM process execution).

As the VM began to execute application code, work on the Inspector proceeded incrementally along several fronts simultaneously:

- *features on demand*: as the VM became more functional and the concerns of the development team evolved, many more features were added: additional views of internal state, more debugging controls, more user options, etc. These were, and continue to be, demand-driven according to the needs of the project.
- *UI functionality and consistency*: the early window implementations were rewritten for code reuse and standardized around new conventions, the menu system was standardized and extended, Java Look & Feel compliance was added, and more.
- *re-architecting internals*: model/view separation was added, direct interaction among views was replaced by a user event model, change propagation was refined, generalized notion of user selection defined, etc.

Once model/view separation became explicit in the previously monolithic code base, the Inspector sources were incrementally split into two "projects" with distinct concerns:

- **Tele**: responsible for communicating with and managing the VM process, essentially being the keeper of the model of the VM's state at any point during the session.
- **Inspector**: responsible for user interaction, state visualization, and command handling.

Dependence between the two projects eventually became one-way, but remained complex: the `Inspector` project depends directly on many implementation classes from both the `Tele` and VM projects. A subsequent effort to further separate the two by re-engineering around new, well-documented interfaces is only partially complete.

As the Inspector evolved into a heavily used debugger, demand grew for multi-threaded management of the VM process so that the GUI would remain live and in particular so that a user could interrupt ("Pause") a running VM. Concurrent operation is now supported, but the retrofit (over complex, distributed interactions in the reading and modeling of VM state) is incomplete Occasional concurrency problems appear as the VM and Inspector evolve.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search