The Jikes^{TM*}Research Virtual Machine User's Guide v2.0.1

November 14, 2001

* **Jikes** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

Contents

1 Introduction **5**
1.1 Welcome to RVM . . . 5
1.2 About this document . . . 5

2 Installation Guide **6**
2.1 System Prerequisites . . . 6
2.1.1 AIX /PowerPC . . . 6
2.1.2 Linux/IA32 . . . 6
2.1.3 Win32 and Cygwin/IA32 . . . 7
2.2 Installation Overview . . . 7
2.3 Installation Steps . . . 8
2.4 RVM Configurations . . . 10
2.4.1 Adaptive Configurations . . . 11
2.5 Cross Platform Building . . . 12
2.6 Building the libraries . . . 13
2.7 Building documentation . . . 13

3 Running RVM **14**
3.1 Running RVM . . . 14
3.1.1 Command-Line Options . . . 14
3.2 Regression Tests . . . 15
3.2.1 Testing Support . . . 15
3.2.2 Individual Test Programs . . . 18
3.2.3 Night Sanity Support . . . 21
3.2.4 An Example: Night Sanity at Watson . . . 21

4 Runtime Implementation Details **23**
4.1 Object Layout . . . 23
4.2 Object headers . . . 23
4.3 Methods and Fields . . . 24
4.4 VM Conventions . . . 26
4.4.1 AIX VM Conventions . . . 26
4.4.2 Lintel VM Conventions . . . 27
4.5 Class Loading . . . 28
4.6 Thread System . . . 29
4.7 VM Callbacks . . . 30

5 Optimizing Compiler Implementation Details **32**
5.1 Options . . . 32
5.1.1 BooleanOptions.dat . . . 32
5.1.2 ValueOptions.dat . . . 32
5.2 Method Compilation . . . 34
5.3 IR Operators . . . 35
5.4 Instruction Formats . . . 35
5.5 BURS Rules . . . 37
5.6 "Magic" Methods . . . 39
5.7 Adaptive Optimization System . . . 39
5.8 OptTestHarness . . . 39

6 Memory Management Details **41**
6.1 Directory Structure . . . 41
6.2 Choosing an Allocator/Collector . . . 41
6.3 Adding A New Allocator/Collector . . . 41
6.4 Load Balancing Work Queue . . . 42
6.5 Generational Write Barrier . . . 42
6.6 Starting Garbage Collection . . . 42
6.7 Measuring Collector Performance . . . 42

7 JDP: the Jikes RVM debugger **44**
7.1 About `jdp` . . . 44
7.2 Getting started . . . 44
7.3 `jdp` environment . . . 45
7.4 `jdp` commands . . . 47
7.5 `jdp` Macros . . . 47
7.6 Debugging tips . . . 48

8 Using the Jikes RVM to Profile an Application **51**
8.1 Profiling An Application . . . 51
8.2 Instrumented Event Counters . . . 51
8.2.1 Existing instrumentation phases . . . 51
8.2.2 Writing new instrumentation phases . . . 52

9 Experimental Guidelines **54**
9.1 Which boot image should I use? . . . 54
9.2 What commmand-line arguments should I use? . . . 54
9.3 RVM is really slow! What am I doing wrong? . . . 54

10 Coding Style Guidelines **56**
10.1 Coding style description . . . 56
10.2 Javadoc requirements . . . 56
10.3 Useful tools/hints . . . 56
10.3.1 emacs . . . 56
10.3.2 vi . . . 57

11 FAQ **58**
11.1 General . . . 58
11.1.1 What is RVM? . . . 58
11.1.2 Who is using RVM? . . . 58
11.1.3 Can I use RVM when teaching a class? . . . 58
11.1.4 Who can I contact with questions? . . . 58
11.1.5 Which mailing list(s) should I subscribe to? . . . 58
11.1.6 How can I contribute to RVM? . . . 59
11.2 Getting RVM and Documentation . . . 59
11.2.1 How do I get RVM? . . . 59
11.2.2 Is there a list of known bugs? . . . 59
11.2.3 Is there documentation on-line? . . . 59
11.2.4 Can I get the Quicksilver Quasi-Static System? . . . 59
11.2.5 Can I get DejaVu? . . . 59
11.3 Building RVM . . . 59
11.3.1 Which jikes should I use? . . . 59
11.3.2 Has anybody thought about incremental boot image writing? . . . 60

11.3.3 Can I recompile the RVM classes without rebuilding the boot image? . . . 60
11.3.4 Can I monitor progress during jbuild? 60
11.3.5 How can I include my own classes in the boot image? 60
11.4 Runtime implementation 60
11.4.1 Does RVM have an interpreter? 60
11.4.2 Does RVM support JNI? 60
11.4.3 Does RVM support user-defined class loaders? 60
11.4.4 Does RVM support the Java security model? 60
11.4.5 Does RVM support serialization? 60
11.4.6 Does RVM enforce the Java Memory Model? 61
11.4.7 Does RVM support zip files? 61
11.4.8 Why doesn't the RVM source use packages? 61
11.4.9 How do RVM's threads, Posix threads, and kernel threads relate to each other? 61
11.4.10 What are the semantics of VM_Uninterruptible? 61
11.4.11 What is the list of operations that may cause a GC? 62
11.4.12 How can I implement a new GC algorithm? 62
11.4.13 How does RVM enter native code? 62
11.4.14 What happens to thread switching while a thread is executing native code? 62
11.4.15 How do the various locking and synchronization mechanisms relate to each other? 62
11.4.16 What causes VM_JNIEnvironment.setNames() to run? 63
11.4.17 Does RVM conform to Sun's JDK Host Porting Interface? 63
11.5 Libraries 63
11.5.1 Does RVM run awt? 63
11.5.2 Can I run some standard library on RVM that is not included in rvmrt.jar? 63
11.6 Optimizing Compiler 63
11.6.1 What is a PEI? 63
11.6.2 What is AOS? 64
11.6.3 Is there a difference between a GC safe point and a thread switch point? . . 64
11.6.4 How do I find the def of a register in SSA form? 64
11.6.5 What is Heap Array SSA form? 64
11.6.6 Is ABCD included? 64
11.6.7 Is escape analysis included? 64
11.6.8 How do I insert my new compiler pass in the optimizing compiler driver? . 64
11.6.9 What if I want my pass to do inter-procedural analysis? 65
11.6.10 What is the OptTestHarness? 65

12 Appendix A: RVM command-line directives 68
12.1 Command-Line Directives in Non-adaptive Configurations 68
12.2 Command-Line Directives in Adaptive Configurations 72
12.2.1 Adaptive Optimization System (AOS) Command-Line Directives 72
12.2.2 Implementing AOS command-line options 75
12.3 Adding RVM Nonstandard Command-Line Options 75

List of Figures

1 Layout of a simple object and an array object in RVM. 24
2 The RVM Table Of Contents and other objects. 25

1 Introduction

1.1 Welcome to RVM

The Jikes[TM1]RVM is a Research Virtual Machine developed at the IBM T.J. Watson Research Center. Key features of RVM include

- the entire virtual machine (VM) is implemented in the Java[TM2]programming language,
- the VM utilizes two compilers and no interpreter,
- a family of parallel, type-exact garbage collectors,
- a lightweight thread package with compiler-supported preemption,
- an aggressive optimizing compiler, and
- a flexible online adaptive compilation infrastructure.

A significant body of information about RVM (formerly known as Jalapeño) appears in our published papers. For overviews of RVM structure, including the runtime system, optimizing compiler, and adaptive systems, see the published papers available from the Jalapeño web page:

http://www.research.ibm.com/jalapeno/publication.html

The best paper for a general introduction to RVM is the IBM Systems Journal, January 2000 paper [1] . For introductions to the optimizing compiler and adaptive system, see the 1999 ACM Java Grande [3] and 2000 OOPSLA [2] papers, respectively.

RVM is a bleeding-edge research project. You will find that some of the code and most of documentation does not live up to product quality standards. Don't hesitate to help rectify this by contributing clean-ups, bug fixes, and missing documentation to the project.

1.2 About this document

This document provides RVM information that is not covered in our published papers. For high-level overviews, algorithms, and structures, you will find the published papers to be the best starting place. This document supplements the RVM distribution, focusing on implementation details of how to build, run, and add functionality to RVM.

This document is available as both postscript and html. You will find the html version more useful, as it includes hyperlinks. The html version is available at `http://www.ibm.com/developerworks/oss/jikesrvm/userguide/HTML/userguide.html`.

The RVM web page includes javadoc API documentation. This html should be the primary reference for individual classes. The javadoc for the optimizing compiler classes contains at least some description for most classes, while most other classes provide javadoc with only a minimal API.

This version of the user's guide is for use with the initial open-source release. You may find sections missing or incomplete. We intend this document to live as a continual work-in-progress, hopefully growing and maturing as community members edit and add to the guide. Please accept this invitation to contribute.

Please send feedback, bug fixes, and text contributions to the RVM researchers mailing list. Constructive criticism will be cheerfully accepted.

---

[1]**Jikes** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

[2]**Java** and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

2 Installation Guide

So you've gotten the RVM distribution. Now what? This section gives instructions on how to install and run RVM for the new RVM user.

2.1 System Prerequisites

To build on any platform you will need the following:

- GNU make. You can download this from `http://www.gnu.org/software/make/make.html`.
- The Jikes Java compiler You can download this from `http://oss.software.ibm.com/developerworks/opensource/jikes`. You can use any of the pre-compiled binaries, or build it yourself from the source. We recommend version 1.13; users have reported difficulties with version 1.14. We have not yet tried version 1.15.
- GNU tar is needed to extract the distribution tar file. You can download this from `http://www.gnu.org/software/tar/tar.html`. We have experienced problems with the AIX™[3]tar program truncating file names.

2.1.1 AIX /PowerPC

In addition to the software mentioned above, to install, build, and run RVM on a AIX/PowerPC environment, you will need

- a PowerPC processor
- AIX 4.3 or later, and
- (recommended) 512MB of memory.
- the IBM AIX Developer Kit for Java version 1.3.0. You can download this from `http://www.ibm.com/java/jdk/aix/index.html`.

2.1.2 Linux/IA32

In addition to the software mentioned above, to install, build, and run RVM on a Linux Intel environment, you will need

- an Intel Architecture 32 bit processor
- Linux (we have run on RedHat 6.0 and 7.0)
- The system has run in 386MB; we have not established a lower bound.
- the IBM Developer Kit (at least version 2.1.3). You can download this from `http://www.ibm.com/java/jdk/aix/index.html`. We did have problems on bleeding edge 2.4 kernels running on SMP machines with versions of the IBM Developer Kit built prior to May 2001. We are currently using IBM build cx130-20010502; later builds should work as well.
- `ksh`. This must be installed in `/bin/ksh`.

[3]**AIX** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

2.1.3 Win32 and Cygwin/IA32

Currently this option for running RVM is barely if at all functional. It is not currently being maintained or tested. You are encouraged to skip this section and don't even think about running RVM on Win32 with cygwin.

If you're still reading, you are probably an enterprising hacker that is interested in getting this configuration up an running. More power to you! We'd like to hear your progress.

In addition to the software mentioned above, to install, build, and run RVM on a Win32/Cygwin Intel environment, you will need

- an Intel Architecture 32 bit processor
- a Win32 operating system (we have only tried it on Windows 2000).
- Cygwin. You can download this from `http://www.cygwin.com`. (I used dll version 1.1.8).
- The system has run in 256MB on Windows 2000; we have not established a lower bound.
- the IBM Developer Kit (at least version 2.1.3). You can download this from `http://www.ibm.com/java/jdk/`.
- `ksh`. One alternative is `pdksh` from `http://www.cs.mun.ca/~michael/pdksh`.

Some issues with this platform include:

- We haven't managed to successfully install a handler for hardware traps, so the first hardware exception (e.g., a `NullPointerException`) will cause an ugly core dump.
- Spaces in directory names aren't handled by jconfigure and the other scripts. Use cygwin's mount command to avoid them (in particular the config file assumes that C:\Program Files\IBM is mounted as /IBM).

2.2 Installation Overview

In order to install and build RVM, you will need to acquire the following items from the RVM Download Page.

- the RVM source distribution. This is available as a compressed tar file `jikesrvm-2.0.1.tar.gz`. You can also work with the contents of this repository with CVS from the public repository.
- the RVM standard libraries. This is a file `jlibraries-2.0.1.tar.gz` available from the download page.

After downloading these files, you will set up a working directory holding the RVM source files, standard library jar, and tools needed to build RVM.

RVM can be configured in various ways. Multiple versions of RVM, corresponding to different configurations, can be generated from one working directory. See Section 2.4 for information about the various configurations. The RVM *boot image* and other files generated during the configuration process are stored in a *build directory* which is logically separate from the working directory.

In order to install RVM you must:

1. Set up a working directory.
2. Set various environment variables.
3. Edit RVM environment scripts.

4. Choose a configuration and run the configuration script to write the appropriate directory and configuration specific files to the build directory.
5. Build an executable version of RVM.

The remainder of this section describes the process in greater detail.

2.3 Installation Steps

1. **Set up a working directory.**

   First extract the RVM source distribution into a directory such as `$HOME/rvmRoot`.

   ```
   % cd $HOME
   % mkdir rvmRoot
   % cd rvmRoot
   % zcat jikesrvm-[version].tar.gz | tar xvf -
   ```

   Next extract the RVM standard libraries. The following installs the standard library file `rvmrt.jar` in `$RVM_ROOT/support/lib`.

   ```
   % cd $HOME/rvmRoot
   % zcat jlibraries-[version].tar.gz | tar xvf -
   ```

2. **Set up environment variables.**

   You need to set up the following shell environment variables:

   `RVM_ROOT` the directory that contains the extracted distribution

   `RVM_BUILD` the directory where you would like the build process to generate an executable RVM configuration

   `RVM_HOST_CONFIG` the configuration file used to specify the software environment on which the system is generated; i.e., where the boot image is generated.

   `RVM_TARGET_CONFIG` the configuration file used to specify the software environment where the system support is generated; i.e., where the "booter" and "C runtime" will be generated.

   `PATH` your path should contain `$RVM_ROOT/rvm/bin` in order to pick up various scripts and utilities

   The following environment variables specify the machine architecture and the operating system on which the system is to run.

   `RVM_TARGET_ARCH` .specifies on what architecture the RVM will run e.g. powerpc, IA32 (for Intel 32 bit)

   `RVM_TARGET_OS` specifies on what Operating System the RVM will run e.g., aix, linux

   Because the ARCH and OS variables default to the system on which the "build" is currently executing, they usually do not need to be explicitly set.

   We recommend you set up these variables in your shell configuration file. For example, for `csh`, you might insert the following into your `.cshrc` file:

   ```
   setenv RVM_ROOT $HOME/rvmRoot       # <--define your working directory
   setenv RVM_BUILD $HOME/rvmBuild     # <--define your current build directory
   setenv PATH $RVM_ROOT/rvm/bin:$PATH
   setenv RVM_HOST_CONFIG $RVM_ROOT/rvm/config/powerpc-ibm-aix4.3.3.0
   setenv RVM_TARGET_CONFIG $RVM_ROOT/rvm/config/powerpc-ibm-aix4.3.3.0
   ```

Note: You should define each of these environment variables as an *absolute* path. The builder template expansion process will crash and burn if you use a `..` in these paths.

For a Linux-Intel environment, the exports would be replaced with the following:

```
setenv RVM_HOST_CONFIG $RVM_ROOT/rvm/config/i686-pc-linux-gnu
setenv RVM_TARGET_CONFIG $RVM_ROOT/rvm/config/i686-pc-linux-gnu
```

For a Cygwin-Intel environment, the exports would be replaced with the following:

```
setenv RVM_HOST_CONFIG $RVM_ROOT/rvm/config/i686-pc-cygwin
setenv RVM_TARGET_CONFIG $RVM_ROOT/rvm/config/i686-pc-cygwin
```

These two variables point to the same file when the type of system doing the build is the same as where you are going the execute the RVM. To cross build a system e.g., build on AIX™[4]/PowerPC™[5] for a Linux/IA32 platform, see the section on Cross Platform Building.

3. **Edit configuration scripts.**

   You must edit a script in the `$RVM_ROOT/rvm/config` directory to set up variables used by the installation process. If someone else at your site has already installed RVM, they have probably already done this step for you. Consult your local RVM guru.

   You must edit the file(s) that define the host and target configuration environments in the `$RVM_ROOT/rvm/config` directory. You do not need to *source* these variables in your working shell; variables in this file will be picked up by the installation scripts. You must set the following variables in this file:

   `HOST_JAVA_HOME` the base directory for JDK JVM

   `HOST_JAVA` the executable command for the JDK JVM

   `HOST_JAVAC` the `javac` executable for the JDK

   `HOST_JAR` the `jar` executable for the JDK

   `HOST_REPOSITORIES` the `rt.jar` archive for the JDK

   `HOST_TOOLS` the `tools.jar` archive for the JDK

   `GNU_MAKE` the GNU `make` executable

   `JIKES` the Jikes compiler executable (`jikes`).

   `CC` how to invoke the C compiler.

   `CPLUS` how to invoke the C++ compiler.

   `LDSHARED` how to link a shared C++ library.

   various basic Unix utilities e.g., `grep`, `xargs`, etc.

   The remaining variables in the config file are not required for basic RVM builds.

   Someday we should consider setting up an autoconf to automate this step.

4. **Choose configuration and populate your build directory.**

   You will use the `jconfigure` script (in `$RVM_ROOT/rvm/bin`) to populate your build (`$RVM_BUILD`) directory with files. You must first choose a RVM configuration.

   For novice users, two configurations are recommended. (A discussion of RVM configurations appears in Section 2.4.)

[4] **AIX** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

[5] **PowerPC** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

- `BaseBaseSemispace`: a non-adaptive system that uses the baseline compiler everywhere, with the semispace copying collector
- `OptOptSemispace`: a non-adaptive system that uses the optimizing compiler everywhere, with the semispace copying collector

Depending on your purposes (See Section 6.2.) you may want to choose another configuration, e.g.,

- `OptOptMarkSweep`: a non-adaptive system that uses the optimizing compiler everywhere, with the mark-sweep (noncopying) collector

Run the `jconfigure` script to set up the `$RVM_BUILD` directory for the configuration you desire. This step creates build scripts for your configuration and otherwise formats your `$RVM_BUILD` directory. The `jconfigure` script takes one argument, the name of the configuration desired:

```
% jconfigure <configuration>
```

For example, to configure a build directory for the `OptOptSemispace` configuration, type the following command:

```
% jconfigure OptOptSemispace
```

5. **Build an executable version of RVM.**

   Use the `jbuild` script, located in the `$RVM_BUILD` directory, to build an executable system. This script copies source files into `$RVM_BUILD/RVM.classes`, preprocesses these files, generates some code with template expansions, builds an executable C program to start the RVM, and writes the RVM boot image. The boot image is the binary image of a ready-to-go instance of the RVM.

   The `jbuild` script must be run from the `$RVM_BUILD` directory. It prints a copious report of its operation which you may save for future reference by redirecting standard out and err.

```
% cd $RVM_BUILD
% jbuild
```

   After the `jbuild` script has completed successfully you should be able to run RVM. (See Section 3.)

   Note: The jbuild process may produce warning messages; these should not affect system viability.

2.4 RVM Configurations

This section describes the RVM build configurations. The various RVM build configurations are defined by files in `$RVM_ROOT/rvm/config/build`.

Most standard RVM configuration files loosely follow the following naming scheme

```
<boot image compiler> <runtime compiler> <garbage collector>
```

where

- the *boot image compiler* is the compiler used to compile the RVM boot image.

- the *runtime compiler* is the "compiler" used to compile the classes loaded at runtime.
- the *garbage collector* is the garbage collection scheme used.

The types of compilers – the baseline compiler and the optimizing compiler – are designated by the names *Base* and *Opt* respectively. In these configurations, all classes loaded at runtime are compiled once, by the specified compiler. This is different than the adaptive configurations, discussed in Section 2.4.1.

A garbage collector may have any of the following types:

NoGC no garbage collection is performed

Semispace a copying semi-space collector

MarkSweep a mark-and-sweep (non copying) collector

CopyGenVariable a copying generational collector with a variable-size nursery

CopyGenFixed a copying generational collector with a fixed-size nursery

HybridGC a hybrid collector, semi-space for the nursery and mark-and-sweep for the mature space

Concurrent a concurrent reference counting collector

For example, to specify a compiler with a baseline-compiled boot image that will compile classes loaded at runtime using the optimizing compiler and that uses a non-generational semi-space copying garbage collector use the name *BaseOptSemispace*.

Some files augment the standard configurations as follows:

- The word *Full* at the beginning of the configuration name identifies a configuration such that all the RVM classes are included in the boot image (by default a small subset of the RVM classes are included in the boot image).
- The word *Fast* at the beginning of the configuration name identifies a Full configuration where all assertion checking has been turned off.

A boot image with either of these modifications is likely to run faster than without (the opt compiler will be opt compiled), but take longer to build.

2.4.1 Adaptive Configurations

In the non-adaptive configurations, all classes loaded at runtime are compiled once by the specified compiler: base or opt. Another option is to build one configuration, an adaptive configuration, where the runtime compiler is either

- specified on the command line (base or opt), or
- selected automatically as the application runs

The first choice allows an adaptive configuration to provide the same functionality as a non-adaptive configuration. One image is built, and the runtime compiler can be specified at the command line as follows:

```
rvm -X:aos:primary_strategy=baseonly
          or
rvm -X:aos:primary_strategy=optonly
```

The second choice initially compiles all methods with the baseline compiler and then automatically selects hot methods for recompilation by the opt compiler at an appropriate optimization level. Further details are provided in Section 5.7.

The adaptive configurations follow the following naming scheme

```
[boot image compiler] Adaptive  <garbage collector>
```

For example, to configure a build directory for an adaptive configuration, where the Opt compiler is used to compile the boot image (but is not included in the boot image and assertions are turned on), and the semi-space garbage collector is used, use the following command:

```
% jconfigure OptAdaptiveSemispace
```

Section 3 describes how this image can be used in the manner mentioned above.

To view a list of configurations see `$RVM_ROOT/rvm/config/build`. Follow the examples in this directory to define your own configurations with different options. See the `jconfigure` file for a list of all options the builder understands.

2.5 Cross Platform Building

The RVM build process consists of two major phases: the building of a RVM *boot image*, and the building of a RVM *boot loader*. The boot image is built using a Java executed within a host JVM and is therefore platform-neutral. By contrast, the boot loader is written in C, and must be compiled on the target platform.

Because the building of the boot image can be a relatively lengthy process, it can be advantageous to perform that task somewhere other than the target platform. By default, the build process will target the operating system and architecture on which the `jconfigure` script is run. Two environment variables can be used to override this behavior and explicitly specify a different target: `RVM_TARGET_ARCH` and `RVM_TARGET_OS` . The current valid target architectures are "powerpc" and "IA32" (intel). The current valid Operating systems are "aix" and "linux".

Two additional environment variables are used to describe the environment on which the boot image is to be built RVM_HOST_CONFIG and where the boot image loader RVM_TARGET_CONFIG is to be built. These variables each point to a file where additional environment variables the location of software components like javac, Jikes , and the C compiler. This distribution contains examples of such files in the `$RVM_ROOT/rvm/config` directory.

For example, to build a BaseBaseSemispace system for AIX™[6] *on a Linux host*:

```
% setenv RVM_ROOT $HOME/rvmRoot
% setenv RVM_BUILD $HOME/rvmBuild
% setenv PATH $RVM_ROOT/rvm/bin:$PATH
% setenv RVM_TARGET_ARCH powerpc
% setenv  RVM_TARGET_OS aix
% setenv RVM_TARGET_CONFIG=$RVM_ROOT/rvm/config/powerpc-ibm-aix4.3.3.0
% setenv RVM_HOST_CONFIG=$RVM_ROOT/rvm/config/i686-pc-linux-gnu
% jconfigure BaseBaseSemispace
% cd $RVM_BUILD
% jbuild
```

This phase of the build process will complete with the words "`please run me on AIX`".

The build process is then completed by building just the boot loader *on an AIX host*:

[6] **AIX** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

```
% setenv RVM_ROOT $HOME/rvmRoot
% setenv RVM_BUILD $HOME/rvmBuild
% setenv PATH $RVM_ROOT/rvm/bin:$PATH
% jbuild -booter
```

After the `jbuild -booter` script has completed successfully you should be able to run RVM.

The building of the boot loader must occur in the same directory as the rest of the build. This can either be done transparently via a network file system, or by copying the build directory from the first host to the target. Of course `RVM_ROOT`, `RVM_BUILD` and `PATH` need not be explicitly set each time: they could have been set in your `.cshrc`.

More advanced users can experiment with the `RVM_BUILD_COPY` environment variable. If this is set, then the `jbuild.linkBooter` phase of the build process is replaced by the execution of `` `$RVM_BUILD_COPY` ``. This opens up a lot of possibilities, including: copying the build directory to a target machine and executing `jbuild.linkBooter` remotely on the target via `rsh` or `ssh`, etc. By setting `RVM_BUILD_COPY` appropriately on the host platform, cross-platform building can become a stream-lined process.

2.6 Building the libraries

Most RVM users will not need to rebuild the libraries; thus the default process described above provides a "binary" `rvmrt.jar`. However, the library source for the RVM is available at `http://www-124.ibm.com/developerworks/projects/jikesrvm`. Please consult the license for restrictions.

Should you decide to modify the library, you will need to rebuild the `rvmrt.jar`. The script to do this is `$RVM_ROOT/rvm/bin/jBuildLibs`. Running this script will compile the library sources in `$RVM_ROOT/support/lib`, and build a new `rvmrt.jar` in `$RVM_BUILD`.

You will find that some versions (including 1.13) of `Jikes` fail to compile the libraries, dying with myriad errors. You need to apply Jikes patch 62 to the Jikes build to fix a problem with variable shadowing by inner classes. We have successfully applied this patch to Jikes version 1.13 on both AIX™[7] and Linux/IA32.

The `jBuildLibs` script will prompt you, asking whether to install the new `rvmrt.jar` in `$RVM_ROOT/support/lib`. If you answer `'y'`, future invocations of `jbuild` will pick up the new library build. However, note that by installing it, you will *overwrite* the original version of `rvmrt.jar`, losing it. Be careful!

2.7 Building documentation

The `jikesrvm-2.0.1.tar.gz` file contains a postscript version of this userguide in `$RVM_ROOT/rvm/doc`. Additionally, the developerWorks web page keeps an on-line version of the userguide and javadoc API, corresponding to the latest HEAD of the CVS repository.

If you would like to recover the userguide or javadoc for an older release of RVM, you can rebuild the documentation locally. See the Makefile in `$RVM_ROOT/rvm/doc/userguide` for rules on how to build the HTML userguide using `hyperlatex`. To build the javadoc pages, use the `jdoc.sh` script in `$RVM_ROOT/rvm/bin`; this script takes as its one command-line argument the directory to output the javadoc HTML.

[7] **AIX** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

3 Running RVM

This section describes how to run an RVM image built from the previous section.

3.1 Running RVM

RVM executes bytecodes from `.class` files. It does *not* compile Java source code. Therefore, all files required by your program must have already been compiled into bytecode files by a Java compiler. We recommend you use the IBM Jikes compiler.

For example, to run class `foo` with source code in file `foo.java`:

```
% jikes foo.java
% rvm foo
```

The general syntax is

```
rvm [rvm options] class [args]
```

You may choose from myriad options for the RVM command-line. Options fall into two categories: *standard* and *non-standard*. Non-standard options are preceded by "-X:".

3.1.1 Command-Line Options

We currently support a subset of the JDK 1.3 standard options. Below is a list of all options and their descriptions. Unless otherwise noted each option is supported in RVM.

-cp -classpath <directories and zip/jar files separated by :> set search path for application classes and resources

-verbose[:gc|:jni|:class] enable verbose output; currently "jni" is not supported

-version print current VM version and terminate

-showversion print current VM version and continue

-? or -help print help message

-X print help on non-standard options

-jar not supported

The non-standard options are

-X:h=<number> allocate <number> megabytes of small object heap

-X:mx<number> allocate <number> megabytes of small object heap

-X:lh=<number> allocate <number> megabytes of large object heap

-X:nh=<number> allocate <number> megabytes of nursery object heap

-X:ph=<number> allocate <number> megabytes of permanent object heap

-X:i=<filename> read boot image from <filename>

-X:sysLogfile=<filename> writes standard error message to <filename>

-X:rvmClasses=<filename> load classes from <filename>

-X:cpuAffinity=<int> physical cpu to which first virtual processor is bound

`-X:processors=<int|''all''>` number of processors to use on a multiprocessor

`-X:verbose` print out additional information for GC

`-X:irc[:help]` print options supported by runtime compiler when in a non-adaptive configuration

`-X:irc<option>` pass `<option>` to the runtime compiler when in a non-adaptive configuration

`-X:aos[:help]` print options supported by adaptive optimization system when in an adaptive configuration

`-X:aos<option>` pass `<option>` to the adaptive optimization system when in an adaptive configuration

See section 12 for more details on command-line options, including the list of options supported by the optimizing compiler and adaptive optimization system.

3.2 Regression Tests

The RVM distribution includes a harness for running regression and performance tests, and several example applications that use this harness. It is located in the rvm/regression subtree in the RVM distribution, and consists of three parts: support for running sanity and performance tests, individual benchmark test subdirectories, and a driver for running nightly sanity tests. The support for running sanity and performance tests comprises a set of scripts and makefiles that define a common interface for running a variety of programs, and for determining their performance and correctness. The individual tests are all located in `$RVM_ROOT/rvm/regression/tests`, and consist of makefiles, scripts sample inputs and outputs; each directory contains support for running an individual application in the testing harness. The support for night sanity runs is a layer on top of actual sanity testing support to ease the task of running and monitoring the tests periodically. Each of the these portions is now described in more detail.

3.2.1 Testing Support

The top-level driver for running sanity tests is called `RunSanityTests`; it is located in the rvm/regression directory and requires `ksh` and *GNU make* in order to run. When it is invoked, it will, by default, build a variety of boot images and run several test programs with each boot image. The behavior of RunSanityTests is highly customizable: most of its behavior can be controlled via command-line options, including what tests and boot images to run, whether to do sanity or performance tests, and what to do with test results. These options are described in more detail below, and a summary can be obtained by invoking `$RVM_ROOT/rvm/regression/RunSanityTests -help`. The behavior of RunSanityTests and the benchmarks must also be customized to your local setup by modifying `$RVM_ROOT/rvm/regression/Local.rules`. In addition, you can also customize by making control files for RunSanityTests to ease running your preferred selection of boot images, test programs and options.

`RunSanityTests` Command-Line Options `RunSanityTests` allows one to specify a variety of options: what tests to run, what boot images to build, how many processors to use, where to put generated boot images and results, whether to run sanity or performance tests, what optimization levels to use for opt compiler tests, and options to pass to the RVM runtime. These command-line options are detailed below:

- `-test` "*test names*"
- `-tests` "*test names*"

- `-test-list` *filename*

  These command line options control what tests to run. The `-test` form expects the next command-line word to be a list of tests (a list of multiple tests must be guarded by quotation marks in most Unix shells); the `-test-list` option takes a single filename as the next argument, and that file must contain a list of tests to run. If the filename is an absolute path, that file is read, if it is a relative path, it is assumed to be relative to $RVM_ROOT/rvm/regression/config. In either case, the items in the list of tests to run must be subdirectories of $RVM_ROOT/rvm/regression/tests (the directory that contains the individual tests).

- `-configuration` "*configuration names*"

- `-configurations` "*configuration names*"

- `-configuration-list` *file name*

  These command line options control what selection of boot images to test. The `-configuration` option expects the next command line word to be a list of boot image configurations understood by jconfigure (a list of multiple tests must be guarded by quotation marks in most Unix shells); the `-configuration-file` option expects that the next command line word be a filename, and that the named file contain a list of boot image configuration names understood by jconfigure. If the filename is an absolute path, that file is read, if it is a relative path, it is assumed to be relative to $RVM_ROOT/rvm/regression/config.

- `-result` *root directory*

  This command line option controls where output generated by the sanity tests should be put. Programs tend to generate output, and running tests can cause compilation and other operations that generate results. To avoid cluttering up the directories containing the actual tests, one can specify that the results be put elsewhere with this option. The directory specified is actually the root of a tree that will be created to contain the results of the different tests and configurations desired; the structure under the specified root directory will mimic the structure of the regression/tests directory.

- `-images` *directory*

  This command line option controls where the boot images to be tested are built. It expects the next command line word to name a directory in which to place or find boot images. Each such boot image requires a corresponding build directory, and these directories placed in the directory specified with this option. In case the `-nobuild` option is given, then the boot images are not built, but rather their build directories must be found, already built, in the directory specified with this option.

- `-use-opt-levels` "*levels*"

  This command line option controls default optimization levels for boot images that have been built with the optimizing compiler as the runtime compiler but without the adaptive system (optimization control under the adaptive system is more complex, and must be done by supplying options using the `-rc-args` flag). This option expects the next command line word to be a list of opt levels (e.g. O0, O1, O2, O3), and the specified tests and boot images will be run once with each specified opt level provided as a boot argument to the runtime compiler. This option also forces results to be placed in separate directories for each optimization level specified.

- `-numprocs` *n*

  This command line option specifies how many processors (i.e. Jikes RVM virtual processors) are to be used for each test run. For every test that is run, the desired number of processors will be given to Jikes RVM as a boot time command line option. This option is ignored for Jikes RVM boot images compiled without support for multiple virtual processors.

- `-nobuild`

  This command line option instructs RunSanityTests not to build the boot images to be tested; in this case, the boot images must have been built already, and be somewhere where RunSanityTests can find them. See the `-images` option for how to tell RunSanityTests where to find prebuilt images.

- `-norun`

  This command line option instructs RunSanityTests not to actually run any tests. This makes sense mostly when used to build boot images that another invocation of RunSanityTests will use.

- `-noclean`

  This option controls whether a clean is performed immediately before each test is run. Giving this option means a clean is not performed.

- `wait` *filename*

  This options is used to synchronize parallel RunSanityTests invocation that are building and running the same set of boot images. When this option is given, the filename specified is created in a boot image directory when the process of building the image completes. And any attempt to actually run a test will wait for this file to exist for the boot image it will use.

- `-performance` *filename*

  This option tells RunSanityTests to run performance tests rather than the default sanity tests. This invokes each desired test in performance mode and collects the performance results into the file specified with this option. In addition, `RunSanityTests` also produces a summary of the resultant performance data when it finishes running all of the tests.

- `-rc-args` "*rvm arguments*"

  This option specifies a set of arguments to Jikes RVM that are to be passed to each invocation of rvm. This option expects the next command line word to be the list of options, so multiple options must be enclosed in quotations.

- `-config-args` "*rvm arguments*"

  This option specifies a set of arguments to jconfigure that are to be passed to jconfigure for each creation of a boot image. This option expects the next command line word to be the list of options, so multiple options must be enclosed in quotations.

`Local.rules` Configuration File The `Local.rules` file is located in the $RVM_ROOT/regression/ directory and is used to customize the testing harness to your local setup. Essentially, it contains the information needed to find test programs not provided as part of the Jikes RVM distribution, such as SPECjvm98 and SPECjbb2000 . Each test harness provided as part of the Jikes RVM for which the benchmark itself is not part of the distribution will have an entry in this file telling the test harness where to find the program itself. In order to use the test harness for those benchmarks, you must obtain the benchmark, install it at your site and edit your `Local.rules` file to reflect the location of each benchmark. `Local.rules` defines the following variables for a benchmark.

BENCH_HOME is the home directory where the benchmark is located. This used for benchmarks for which a test harness but not the benchmark itself is distributed with the Jikes RVM distribution.

BENCH_JARS defines a set of jar files to be included in the classpath of compiles and runs of this benchmark.

Customized Control Files `RunSanityTests` can be given two kinds of control files to make governing its behavior easier: lists of boot images and lists of tests. With the `-configuration-list` option, you can provide a file that lists the set of boot images to build. This file is a plain text file, with one boot image name per line; each boot image name must be one that is understood by jconfigure. With the `-test-list` options, you can provide a filename that the set of tests to run. Similar to the configuration list file, this file is plain text, and has one test name per line. Both types of file should be placed in the $RVM_ROOT/rvm/regression/config directory; when placed there, you can refer to them with an unqualified file name on the RunSanityTests command line. You can use files from other places as control files, but those require an absolute path (i.e. one beginning with a /) on the command line.

3.2.2 Individual Test Programs

The `tests` subdirectory of the `$RVM_ROOT/rvm/regression` directory contains a set of test programs, one in each directory. These directories do not contain the actual program, but rather the scripts and makefiles required to run it in the test harness. The one essential file is `Makefile`, which is makefile written with the GNU make syntax. The structure of these files is as follows; the sections will be discussed below.

1. Include the environment rules from the current build directory
2. Set the benchmark name
3. Include local configuration information, if needed
4. Define benchmark-specific parameters
5. Include the test harness makefile
6. Define benchmark-specific rules

Include the environment rules from the current build directory This is simply the include statement given below. It is required so that subsequent portions of the makefile can find the Jikes RVM itself and information needed from the boot image directory. This section always the one line shown below.

```
include         $(RVM_BUILD)/Make.rules
```

Set the benchmark name This is the name by which the benchmark is identified in the local configuration file, and this name is also used in some situations by `RunSanityTests` for identifying output. It is also possible to specify inclusion of other benchmarks, to pick up any local definitions they may have. This section can define the following make variables.

BENCH_NAME sets the name of this benchmark.

BENCH_INCLUDE is a list of benchmark names to include in this one. This means, for instance, that extra jar files specified for these benchmarks will also be included in this one.

Include local configuration information, if needed This is where any information needed to customize the benchmark to your local configuration is read. It must occur after `BENCH_NAME` is set because the information stored in the local configuration file is kept in terms of benchmark names. This section must always be the following single line.

```
include $(JAL_ROOT)/rvm/regression/Local.rules
```

Define benchmark-specific parameters This defines essential benchmark information such as the name of the class with which to start up, how much heap memory to use, what classpath to use, and the like. There are many variables that the test harness understands, and they can be grouped into three categories: general configuration, sanity checking and benchmark invocation.

The general setup options control source and class paths, the heap sizes needed by the benchmark and any extra boot time arguments to RVM. These variables are as follows:

BENCH_SOURCE_PATH is a list of directories in which to find Java source files that are used by this benchmark. This is not a colon-separated classpath, but a normal whitespace-separated makefile list.

BENCH_CLASS_PATH is a list of directories and jar files in which to find Java class files that are used by this benchmark. This is not a colon-separated classpath, but a normal whitespace-separated makefile list.

HEAPSIZE is the amount heap space to use for the normal heap

LHEAPSIZE is the amount heap space to use for the large object space.

BENCH_RVM_ARGS is extra boot time arguments to provide to RVM.

Sanity checking variables provide a way to specify how to compare the result of running the program to see if it correct. You can define an expected output and also filters to remove extraneous program output before comparison. These variables are as follows:

EXPECTED is a file containing expected output of a program, against which to diff to check whether the program generated the correct result.

AWK_FILTER is a filter to apply to the output of the program before diffing with EXPECTED. This filter will be applied by GNU awk. Defining both AWK_FILTER and SED_FILTER is not allowed.

SED_FILTER is a filter to apply to the output of the program before diffing with EXPECTED. This filter will be applied by GNU sed. Defining both AWK_FILTER and SED_FILTER is not allowed.

The benchmark command line variables control the Java class to start and any arguments to pass to it; there are two distinct sets of these variables depending upon whether the benchmark is a stand-alone program or a client-server one. For stand-alone programs, the variables are as follows:

START_CLASS is the name of the class to give when invoking RVM for this benchmark; this is a normal class name, not a descriptor, so it expects dots rather than slashes for package names.

START_ARGS is a string of arguments to pass when invoking RVM for this benchmark. These arguments are passed after the class name, so they are arguments to the Java program itself, not boot time arguments to RVM.

The client-server programs require more variables since they must start both client and server for a test. In fact, there are three sets of variables: one used to start the server, one used to start the client, and one used to stop the server. The client presumably will stop by itself, but servers typically must be started and stopped explicitly. The variables are as follows:

- **SERVER_NAME** is the name of the main Java class to use when starting the server program.
- **SERVER_ARGS** is a string of the arguments to pass to the server program when it is started.
- **SENTINEL** is output from the server program which indicates that it is ok to start the client program.
- **CLIENT_NAME** is the name of the main Java class to use when starting the client program.
- **CLIENT_ARGS** is a string of the arguments to pass to the client program when it is started.
- **STOP_NAME** is the name of the main Java class to use when stopping the server program.
- **STOP_ARGS** is a string of the arguments to pass to the program that stops the server.

Include the test harness makefile This section loads the rules required to actually compile any Java files that need it, run the program, check the output for correctness and assess performance. This section must always have the following single line:

```
include $(JAL_ROOT)/rvm/regression/Make.rules
```

Define benchmark-specific rules This section defines the rules needed for the specific benchmark; these rules will utilize the rules from the test harness. The benchmark needs to define at least the following rules:

- **sanity** This rule defines what happens when someone types `make sanity`. Typically, it will have one dependence, which is the type of sanity test to perform, of which there are currently the following four. All four rules run the program and capture the output; they differ in how they check for correctness:
  - **sanity-diff** uses the EXPECTED file to check correctness. This is only supported for stand-alone programs.
  - **sanity-compare** runs the same program with the JDK and compares the results with those obtained from RVM. This is only supported for stand-alone programs.
  - **sanity-check-rule** invokes the "check" make rule, which must be defined by the test program makefile. The "check" rule can assume that the variable OUT denotes the output file.
  - **server-sanity-check-rules** is like "check" but for client-server programs. It invokes the "client-check" rule with the client output and the "server-check" rule with the server output.
- **do-gather-performance** This optional rule is used in performance mode. It is given the output file of the program in OUT, and is expected to write arbitrary text describing performance to the file PERF_LOG. One line of this output should be a summary and that line must start with the phrase "Bottom Line:"

For complex benchmarks, you may need other rules. One common situation is that badly-written programs (such as the spec benchmarks) often do not provide the ability to tell them where to look for input files; the files simply must be in the working directory. In that case, you will need rules to copy any needed input files into the directory WORKING, where the program is actually run. See the spec benchmark makefiles for examples of this. Another case we have encountered is needing explicit dependencies telling the test harness that some .class files are located in a .jar file.

3.2.3 Night Sanity Support

One common use of regression tests is running them periodically to keep track of system state over time; the Jikes RVM test harness provides a `NightSanityDriver` to simplify this process. This script is a wrapper around `RunSanityTests` that takes a configuration file of boot images and testing options, and uses that to drive running sanity tests. `NightSanityDriver` takes two command line arguments:

-common "*options*" is a list of options to apply to all invocations of `RunSanityTests`. This is optional.

-config *filename* is a configuration file that specifies a series of invocations of `RunSanityTests`. If this filename is relative, it is assumed to be in $RVM_ROOT/rvm/regression/config; if it is absolute, it is used as it is given. The format of this file is discussed below.

Given these arguments, `NightSanityDriver` invokes `RunSanityTests` as instructed on each line of the config file, appending the common options, if any.

The `NightSanityDriver` Config File The config file for `NightSanityDriver` specifies a series of invocations of `RunSanityTests`, one per line of the file. Each line starts with a boot image name, and the rest of the line is options to pass to `RunSanityTests`. `RunSanityTests` is invoked to run just that boot image with the options specified.

3.2.4 An Example: Night Sanity at Watson

To illustrate how the various control files for `RunSanityTests` and `NightSanityDriver` can be used, we include a simplified version of the night sanity configuration used at Watson. All configuration files discussed go in the $RVM_ROOT/rvm/regression/config directory. Since it is too time consuming to test every possible variation of Jikes RVM on every program, we have selected a representative sample to run every night. The first level is to define two sets of benchmark programs, a full one and small one used for configurations considered less critical. Thus, we have the following two files:

nightly-tests file

```
bytecodeTests
opttests
reflect
threads
utf8
CaffeineMark
jBYTEmark
javalex
SPECjvm98
SPECjbb2000
```

nightly-tests-short file

```
bytecodeTests
opttests
reflect
threads
utf8
CaffeineMark
jBYTEmark
```

We then have a control file for `NightSanityDriver` that runs a selection of boot images using these two sets of regression tests. Note that we use the nightly-tests-short list of tests for configurations that mainly for debugging purposes (ExtremeAssertionsOptOptSemispace) or are configurations that one can create but would not want to use in practice (BaseAdaptiveSemispace). To ensure relatively complete coverage testing of the optimizing compiler, we chose one optimizing configuration (FullOptMarkSweep) on which to run with all the different optimization levels.

```
BaseBaseSemispace -test-list nightly-tests
BaseBaseMarkSweep -test-list nightly-tests
BaseBaseCopyGenVariable -test-list nightly-tests
BaseAdaptiveSemispace -test-list nightly-tests-short
FullOptMarkSweep -use-opt-levels "O0 O1 O2" -test-list nightly-tests
FullOptCopyGenVariable -test-list nightly-tests
FullAdaptiveSemispace -test-list nightly-tests
FastSemispace -test-list nightly-tests
FastMarkSweep -test-list nightly-tests
FastAdaptiveSemispace -test-list nightly-tests
ExtremeAssertionsOptOptSemispace -test-list nightly-tests-short
```

To make all of this run every night, it is necessary merely to put a NightSanityDriver invocation in a crontab; we actually use a more involved setup, which we distribute in the night-sanity-build and night-sanity-run scripts. These are somewhat specific to what we want to test, but they will hopefully make a good model.

4 Runtime Implementation Details

This section provides some information on various implementation details for the RVM runtime system.

4.1 Object Layout

Values in the Java programm language are either *primitive* (e.g. `int`, `double`, etc.) or they are *references* (that is, pointers) to objects. Objects are either *arrays* having components or *simple objects* having fields. RVM's object model is governed by four criteria:

- field and array accesses should be as fast as possible,
- null-pointer checks should be performed by the hardware if possible,
- virtual method dispatch should be fast, and
- other (less frequent) Java operations should not be prohibitively slow.

Assuming the reference to an object is in a register, the object's fields can be accessed at a fixed displacement in a single instruction. To facilitate array access, the reference to an array points to the first (zeroth) component of an array and the remaining components are laid out in ascending order. The number of components in an array, its *length*, is kept just before its first component.

The Java programming language requires that an attempt to access an object through a `null` object reference generate a NullPointerException. In the RVM, references are machine addresses, and `null` is represented by address 0. The AIX™[8]operating system permits loads from low memory, but accesses to very high memory, at small *negative* offsets from a null pointer, normally cause hardware interrupts.[9] Thus, attempts to index off a null array reference are trapped by the hardware, because array accesses require loading the array length which is -4 bytes off the array reference. A hardware null-pointer check for field accesses is effected by locating fields at negative offsets from the object reference.

In summary, in RVM, arrays grow up from the object reference (with the array length at a fixed negative offset), while simple objects grow down from the object reference with all fields at a negative offset (see figure 1). A field access is accomplished with a single instruction using base-displacement addressing. Most array accesses require three instructions. A single trap instruction verifies that the index is within the bounds of the array. Except for `byte` (and `boolean`) arrays, the component index must then be shifted to get a byte index. The access itself is accomplished using base-index addressing.

4.2 Object headers

A two-word object header is associated with each object. This header supports virtual method dispatch, dynamic type checking, memory management, synchronization, and hashing. It is located 12 bytes below the value of a reference to the object. (This leaves room for the length field in case the object is an array, see figure 1.)

One word of the header is a *status* word. The status word is divided into three *bit-fields*. The first bit-field is used for locking. The second bit-field holds the default hash value of hashed objects.

[8] **AIX** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

[9] In AIX, it is at least theoretically possible for another process to cause a shared system library to get loaded into very high memory. This remote possibility is not a concern in a research project, but would need to be addressed by a commercial JVM. It would be sufficient to forbid read and write access to the last page of addressable memory. (Accesses to some of the fields of objects bigger than a page could be checked explicitly without having a major impact on performance.)

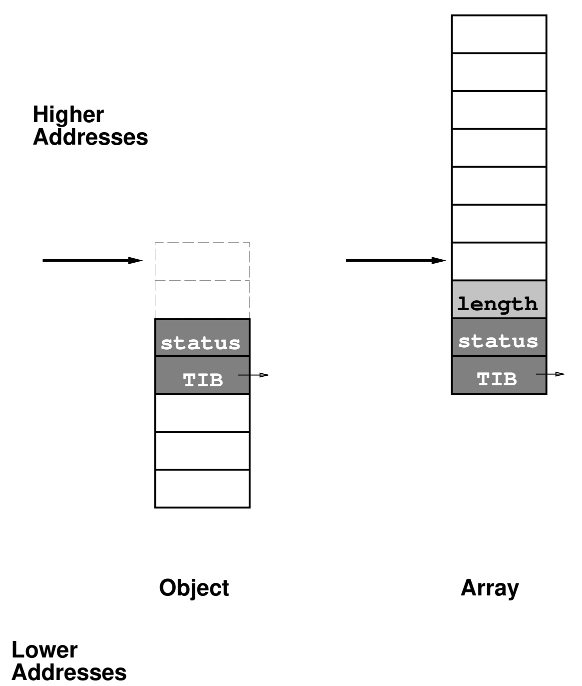

Figure 1: Layout of a simple object and an array object in RVM.

The third bit-field is used by the memory management subsystem. (The size of these bit-fields is determined by build-time constants.)

The other word of an object header is a reference to the *Type Information Block* (TIB) for the object's class. This structure describes the object's class including its superclass and the interfaces it implements as well as has pointers to the class' virtual methods.

4.3 Methods and Fields

A compiled method body is an array of machine instructions (stored as `int`s). While the pointers for static fields and methods are stored in the *JikesRVM Table of Contents* (JTOC), pointers for instance fields and virtual methods are stored in the class's TIB. Consequently the dispatch mechanism is different for static and virtual methods.

The JTOC All of RVM's global data structures are stored in the JTOC. Literals, numeric constants and references to String constants, are also stored there. To enable fast common-case dynamic type checking, the JTOC also contains references to the TIB for each class in the system. Since these structures can have many types and the JTOC is declared to be an array of `int`s RVM uses a descriptor array, co-indexed with the JTOC, to identify the entries containing references. A reference to the JTOC is maintained in a dedicated machine register (the JTOC register). The JTOC is depicted in figure 2.

Virtual Methods A TIB contains pointers to the compiled method bodies (executable code) for the virtual methods of its class. Thus, the TIB serves as RVM's virtual method table. A virtual method dispatch entails loading the TIB pointer at a fixed offset off the object reference, loading the address of the method body at a given offset off the TIB pointer, moving this address to the

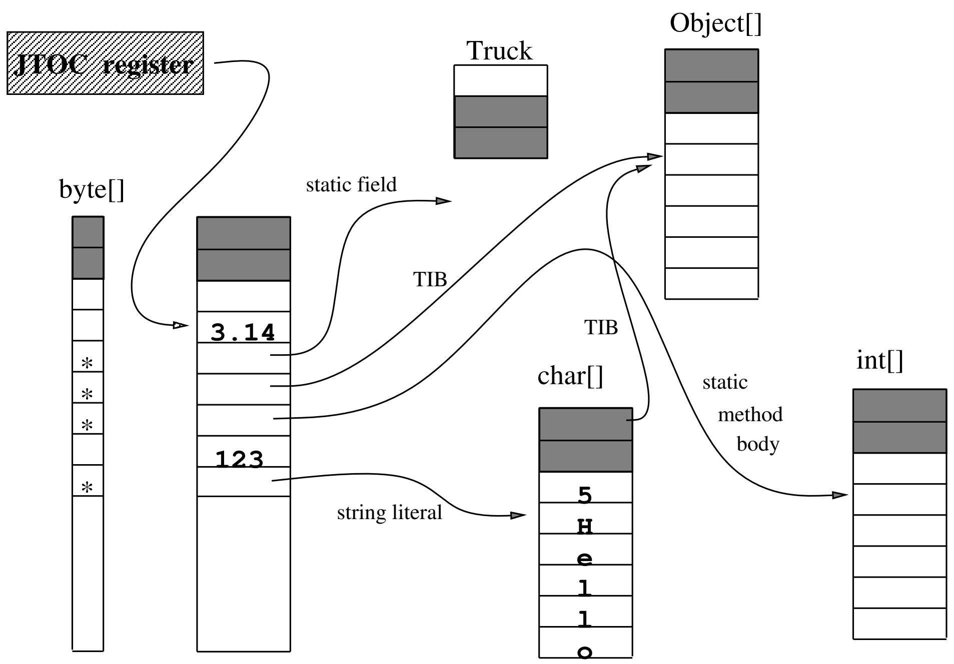

Figure 2: The RVM Table Of Contents and other objects.

PowerPC^{TM}[10] "link-register", and executing a branch-and-link — four instructions.

Static Fields and Methods Static fields and methods are stored in the JTOC. Static method dispatch is simpler than virtual dispatch requiring only that the offset of method in the JTOC be read to find the address of the method.

Lazy Method Compilation The slots in the TIB or the JTOC may be filled in with a pointer to the compiled code for the method itself or if lazy method compilation is enabled and the method has not yet been compiled it may be filled in with a pointer to the compiled code of the *lazy method invocation stub*. If the lazy method invocation stub is invoked its action is to compile the method, substitute a pointer to the compiled code of the method in the slot in the TIB or the JTOC from which it was invoked and then cause execution to jump to the start of the compiled method.

Interface Methods Regardless of whether or not a method is overridden in a class that inherits it virtual method dispatch is still very simple since the method body will be at the same offset in the TIB in its defining class and in every class that inherits from it. However, where the method is an interface method, that is where it is invoked through an `invoke_interface` call rather than an `invoke_virtual call`, its offset is not the same for every class that implements its interface and dispatch is more difficult. The simplest, and least efficient way, of locating an interface method is to search all the virtual method entries in the TIB until a match is found. Another way uses an *Interface Method Table* (IMT) which is much like the TIB. Any method that could be an interface method has a fixed offset into the IMT just as with the TIB. However, unlike in the TIB, two different methods may share the same offset into the IMT. In this case, a *conflict resolution stub* is

[10] **PowerPC** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

inserted in the IMT. Conflict resolution stubs are custom-generated machine code sequences that test the value of a hidden parameter to dispatch to the desired interface method.

4.4 VM Conventions

4.4.1 AIX VM Conventions

This section describes register, stack, and calling conventions that apply to RVM on PowerPC™[11].

Stackframe layout and calling conventions may evolve as our understanding of the RVM's performance improves. Where possible API's should be used to protect code against such changes. In particular, we may move to the AIX conventions at a later date. Where code differs from the AIX conventions, it should be marked with a comment to that effect containing the string "AIX".

Register conventions

Registers (general purpose, gp, and floating point, fp) can be roughly categorized into four types:

Scratch Needed for method prologue/epilogue. Can be used by compiler between calls.

Dedicated Reserved registers with known contents:

- **JTOC - JikesRVM Table Of Contents** Globally accessible data: constants, static fields and methods.
- **FP - Frame Pointer** Current stack frame (thread specific).
- **TI - Thread (locking) Id** Used to set (and test) the locking field of light weight object locks. Can also be shifted to get the index of an object representing the current thread in into a global array.
- **PR - Processor register** An object representing the current virtual processor (the one executing on the CPU containing these registers). A field in this object contains a reference to the object representing the VM_Thread being executed.

volatile ("caller save", or "parameter") Like scratch registers these can be used by the compiler as temporaries, but they are not preserved across calls. (Volatile registers differ from scratch registers in that they (the former) can be used to pass parameters and result(s) to and from methods.)

Nonvolatile ("callee save", or "preserved") These can be used (and are preserved across calls), but they must be saved on method entry and restored at method exit. Highest numbered registers are to be used first. (At least initially, nonvolatile registers will not be used to pass parameters.)

Condition Register's 4-bit fields CR0 - CR1 scratch

- **CR2 - TSCR** dedicated (thread switching, bit 8 TSCRB) (this convention is being phased out, and in fact is not being used in RVM 2.0)
- **CR3 - CR7** scratch

Stack conventions

Stacks grow from high memory to low memory. The layout of the stackframe appears in a block comment in `$RVM_ROOT/rvm/src/vm/arch/powerpc/VM_StackframeLayoutConstants.java`.

Calling Conventions

[11] **PowerPC** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

Parameters All parameters (that fit) are passed in VOLATILE registers. Object reference and int parameters (or results) consume one GP register; long parameters, two gp registers (low-order half in the first); float and double parameters, one fp registers. Parameters are assigned to registers starting with the lowest volatile register through the highest volatile register to the highest nonvolatile of the required kind (gp or fp).

Any additional parameters are passed on the stack in an parameter spill area of the caller's stack frame. The first spilled parameter occupies the lowest memory slot. Slots are filled in the order that parameters are spilled.

An int, or object reference, result is returned in the first volatile gp register; a float or double result is returned in the first volatile fp register; a long result is returned in the first two volatile gp registers (low-order half in the first);

Method prologue responsibilities (some of these can be omitted for leaf methods):

1. Save the caller's next instruction pointer (callee's return address, from the Link Register).
2. Save any nonvolatile floating-point registers used by callee.
3. Save any nonvolatile general-purpose registers used by callee.
4. Store and update the frame pointer FP.
5. Store callee's compiled method ID (eventually, this may not be needed).
6. Check to see if the Java thread must yield the VM_Processor (and yield if threadswitch was requested).

Method epilogue responsibilities

1. Restore FP to point to caller's stack frame.
2. Restore any nonvolatile general-purpose registers used by callee.
3. Restore any nonvolatile floating-point registers used by callee.
4. Branch to the return address in caller.

4.4.2 Lintel VM Conventions

This section describes register, stack, and calling conventions that apply to RVM on Linux/IA32. *Linux/IA32 conventions are still changing; be sure to check the relevant files for the most accurate information!*

Register conventions

EAX First GPR parameter register, first GPR result value (high-order part of a long result), otherwise volatile (caller-save).

ECX Scratch.

EDX Second GPR parameter register, second GPR result value (low-order part of a long result), otherwise volatile (caller-save).

EBX Nonvolatile.

ESP Stack pointer.

EBP Frame pointer, valid at method call/return, must be shadowed in the framePointer field of the VM_Processor object pointed to by ESI (see below). Expected to become a nonvolatile (and FP only in the baseline compiler) very soon.

ESI Processor register, reference to the VM_Processor object for the current virtual processor.

EDI Nonvolatile. (JTOC in baseline compiler)

Stack conventions

Stacks grow from high memory to low memory. The layout of the stackframe appears in a block comment in `$RVM_ROOT/rvm/src/vm/arch/intel/VM_StackframeLayoutConstants.java`.

Calling Conventions

At the beginning of callee's prologue The first two areas of the callee's stackframe (see above) have been established. ESP points to caller's return address. Parameters from caller to callee are as mandated by `$RVM_ROOT/rvm/src/vm/arch/intel/VM_RegisterConstants.java`.

After callee's epilogue Callee's stackframe has been removed. ESP points to the word above where callee's frame was. EBP points to A's frame. (The framePointer field of the VM_Processor object pointed to by ESI must also contain this value.) If B returns a floating-point result, this is at the top of the fp register stack. If B returns a long, the low-order word is in EAX and the high-order word is in EDX. Otherwise, if B has a result, it is in EAX.

4.5 Class Loading

RVM implements the Java programming language's dynamic class loading. While a class is being loaded it can be in one of five states. These are

vacant a forward reference exists to the class but loading has not yet begun.

loaded the class's bytecode file has been read and parsed successfully.

resolved the superclass of this class has been loaded and resolved and the offsets (whether in the object itself, the JTOC, or the class's TIB) of its fields and methods have been calculated.

instantiated the superclass has been instantiated and pointers to the compiled methods have been inserted into the JTOC(for static methods) and the TIB (for virtual methods).

initializing the superclass has been initialized and the class initializer is being run.

initialized the superclass has been initialized and the class initializer has been run.

The class passes through these states in the following fashion.

Vacant The `VM_Class` object for this class has been created and registered. A class can be in this state if a reference to the class exists in a constant pool of some other class.

Loaded In this state the class file has been read and parsed. The constant pool has been constructed. The declared methods and fields of the class have been loaded. Loading a method or field consists of reading its modifiers and attributes.

Resolved In this state the superclass of this class has been loaded and resolved. A list of the virtual methods and instance fields of this class, including the methods and fields inherited from its superclass has been constructed and the offsets for the instance fields have been calculated. Space has been allocated in the JTOC for all static fields of the class and for static method pointers and the appropriate offsets calculated. The TIB has been initialized and offsets for the virtual methods have been calculated.

Instantiated In this state the superclass of this state has been instantiated. The slots in the TIB are filled in with pointers to the compiled code for the virtual methods. The slots in the JTOC are filled in with pointers to the compiled code for the static methods.

Initializing In this state the superclass has been initialized. The class initializer is being run.

Initialized In this state the superclass has been initialized. The class initializer has been run.

4.6 Thread System

This section provides some explanation of how Java threads are scheduled and synchronized by the RVM.

All Java threads (application threads, garbage collector threads, *etc.*) derive from `VM_Thread`. These threads are multiplexed by one or more virtual processors (see `VM_Processor`). Normally, the number of RVM virtual processors to use is a command line argument (e.g. `-X:processors=4`) Generally, there should be one RVM virtual processor for each CPU on an SMP. Additional virtual processors may be created to handle threads executing non Java code through the Java JNI. Multiple virtual processors require a working pThread library, each virtual processor being bound to a pThread. It is possible to build a system that only uses one virtual processor by setting the preprocessor directive `JVM_WITH_SINGLE_VIRTUAL_PROCESSOR` to 1. This may give a minor performance benefit on uniprocessors.

(Bugs with our Linux pThread library currently prohibit multiple virtual processor builds on Linux. Code to support such builds is in place, but has not been tested.)

Threads that are not executing are either placed on thead queues (deriving from `VM_AbstractThreadQueue`) or are proxied (see below). Thread queues are either global or (virtual) processor local. The latter do not require synchronized access but global queues do. Unfortunately, we did not see how to use Java monitors to provide this synchronization. (In part, because it is needed to implement monitors, see below.) Instead this low-level synchronization is provided by `VM_ProcessorLock`s.

Transferring execution from one thread (A) to another (B) is a complex operation negotiated by the `yield` and `morph` methods of VM_Thread and the `dispatch` method of VM_Processor. `yield` places A on an indicated queue (releasing the lock on the queue, if it is global). `morph` uses `VM_Magic` to capture the state of the running thread and transfers control to `dispatch`. At this point, the virtual processor is executing in *phantom mode*, it is using A's stack, but not in a way that will be visible to A when it next gets executed. `dispatch` removes B from a queue of executable threads and, using more VM_Magic, transfers control to B's stack. (To B, it looks as if *its* call to `dispatch` has just returned.) To prevent a different processor from dispatching A while it is still executing in phantom mode, `yield` sets the `beingDispatched` field of A, which is only reset by the magic that transfers control to B.

Currently, RVM has no priority mechanism, that is, all threads run at the same priority.

Similarly, it has only the most rudimentary load balancing mechanism. Each virtual processor has a (local) idle queue. Normally, the processor's `VM_IdleThread` inhabits this queue. When no other thread is executable, this thread is executed. This thread requests work by setting the static `idleProcessor` field to its virtual processor. (When `dispatch` is looking for a runnable thread it checks that this field is null. If not, and it has an spare runnable thread, it places the spare thread on the idle processor's transfer queue.) The idle thread then spins for a short period of time waiting for work to materialize. If it does, the idle thread returns to its processor's idle queue and the processor resumes execution of the newly transferred thread. Otherwise, the idle thread surrenders the remainder of its virtual processor's time slice back to the operating system.

More sophisticated priority and load-balancing mechanisms are in order.

RVM uses a light-weight locking scheme to implement Java monitors (see `VM_Lock`). Twenty bits of the status word of the object header are used for locking. If the top bit is set, the bottom nineteen are an index into an array of heavy-weight locks. Otherwise, if the object is locked, these bits contain the id of the thread that holds the lock and a count of how many times it is held. If a thread tries to lock an object locked with a light-weight lock by another thread, it can spin, yield, or inflate the lock. Spinning is probably a bad idea. The number of times to yield before inflating is a matter open for investigation (as are a number of locking issues, see `VM_Lock`). Heavy-weight locks contain an `enteringQueue` for threads trying to acquire the lock.

A similar mechanism is used to implement Java wait/notification semantics. Heavy-weight locks contain a `waitingQueue` for threads blocked at a Java `wait`. When a `notify` is received, a thread is taken from this queue and transferred to a ready queue. Priority `wakeupQueue`s are used to implement Java sleep semantics. Logically, Java timed-wait semantics entail placing a thread on both a `waitingQueue` and a `wakeupQueue`. However, our implementation only allows a thread to be on one thread queue at a time. To accommodate timed-waits, both `waitingQueue`s and `wakeupQueue`s are queues of *proxies* rather than threads. A `VM_Proxy` can represent the same thread on more than one proxy queue.

4.7 VM Callbacks

The RVM provides callbacks for many runtime events of interest to the RVM programmer, such as classloading, VM bootimage creation, and VM exit. The callbacks allow arbitrary code to be executed on any of the supported events.

The callbacks are accessed through the nested interfaces defined in the `VM_Callbacks` class. There is one interface per event type. To be notified of an event, register an instance of a class that implements the corresponding interface with `VM_Callbacks` by calling the corresponding `add...()` method. For example, to be notified when a class is instantiated (see section 4.5), first implement the `VM_Callbacks.ClassInstantiatedMonitor` interface, and then call `VM_Callbacks.addClassInstantiatedMonitor()` with an instance of your class. When any class is instantiated, the `notifyClassInstantiated` method in your instance will be invoked.

The RVM currently supports callbacks for the following events.

- `ClassLoaded`, which happens when a class is *loaded*.
- `ClassResolved`, which happens when a class is *resolved*.
- `ClassInstantiated`, which happens when a class is *instantiated*.
- `ClassInitialized`, which happens when a class is *initialized*.
- `MethodOverride`, which happens when a method in a newly loaded class overrides a method in an existing class.
- `ForName`, which happens when java.lang.Class.forName() is invoked.
- `BootImageWriting`, which happens when boot image writing is started.
- `Exit`, which happens when the RVM is about to exit.

The appropriate interface names can be obtained by appending “Monitor” to the event names (e.g. the interface to implement for the `MethodOverride` event is `VM_Callbacks.MethodOverrideMonitor`). Likewise, the method to register the callback is “add”, followed by the name of the interface (e.g. the register method for the above interface is `VM_Callbacks.addMethodOverrideMonitor()`).

NOTE: there is currently no `Startup` event. It is in the process of being implemented.

Since the events for which callbacks are available are internal to the RVM, there are naturally some limitations on the behavior of the callback code. For example, as soon as the exit callback is invoked, all threads are considered daemon threads (i.e. the VM will not wait for any new threads created in the callbacks to complete before exiting). Thus, if the exit callback creates any threads, it has to `join()` with them before returning. These limitations may also produce some unexpected behavior. For example, while there is an elementary safeguard on any classloading callback that prevents recursive invocation (i.e. if the callback code itself causes classloading), there is no such safeguard across events, so, if there are callbacks registered for both `ClassLoaded` and `ClassInstantiated` events, and the `ClassInstantiated` callback code causes dynamic class loading, the `ClassLoaded` callback will be invoked for the new class, but not the `ClassInstantiated` callback.

Examples of callback use can be seen in the `VM_Controller` class in the adaptive system and most `VM_Allocator` classes.

5 Optimizing Compiler Implementation Details

This section provides some information on various implementation details for the RVM optimizing compiler.

5.1 Options

The command-line options to the optimizing compiler are stored as fields in an object of type `OPT_Options`. The RVM build process generates the `OPT_Options.java` file automatically from a template.

To add or modify the command-line options in `OPT_Options.java`, you must modify either `BooleanOptions.dat` or `ValueOptions.dat`. You should describe your desired command-line option in a format described below. Your option will be generated the next time you build the system.

5.1.1 BooleanOptions.dat

The `BooleanOptions.dat` file defines boolean options for the optimizing compiler. Each command-line option is described by a two-line record, and each record is separated by a blank line. Long lines can be partitioned using "\". **NOTE:** blank lines *are* important! Lines starting with "#" are ignored.

The first line must have the following format

```
FULL_NAME OPT_LEVEL DEFAULT_VALUE {SHORT_NAME}
```

where

- `FULL_NAME` gives the name of the boolean field in `OPT_Options.java`
- `OPT_LEVEL` gives the minimum optimization level that automatically sets this field true
- `DEFAULT_VALUE` of `true` or `false`
- `SHORT_NAME` is an optional field which defines a mnemonic by which the command-line processor recognizes this option.

The second line of each record must be a short textual description of the semantics of the option. This description will be printed by `-X:irc:help` (see Section 12).

For example, the two line record in `BooleanOptions.dat` that defines the option of whether to perform local scalar replacement is

```
LOCAL_SCALAR_REPLACEMENT 1 true local_sr
Perform local scalar replacement
```

5.1.2 ValueOptions.dat

The `ValueOptions.dat` file defines non-boolean options for the optimizing compiler. Each command-line option is described by a three-line record, and each record is separated by a blank line. As with `BooleanOptions.dat`, long lines can be broken by using "\" and blank lines are once again significant. Lines starting with "#" are ignored.

The first line must have the following format

```
TAG FULL_NAME TYPE DEFAULT_VALUE {SHORT_NAME}
```

where

- `TAG` is 'E' for an Enumeration type, and 'V' for a value type. Further instructions for Enumeration types appear below.
- `FULL_NAME` gives the name of the field in `OPT_Options.java`
- `TYPE` is one of 'byte', 'int', or 'String', and gives the primitive datatype for the value in OPT_Options.java
- `DEFAULT_VALUE` is the default value for the option
- `SHORT_NAME` is an optional field which defines a mnemonic by which the command-line processor recognizes this option.

The second line of each record must be a short textual description of the semantics of the option. This description will be printed by `-X:irc:help` (see Section 12).

The third line of each record is used for enumeration options, and must be left blank for other options.

For example, the three-line record in `ValueOptions.dat` that defines the maximum inlining depth when using static inlining heuristics is

```
V IC_MAX_INLINE_DEPTH int 5
Static inlining heuristic: Upper bound on depth of inlining
<blank line>
```

Enumeration options provide a mechanism to define an option in terms of a small fixed set of choices. For an enumeration option, the third line of the record should contain a specification for each value that the enumeration can take. Each such specification must have the following format:

```
"ITEM_NAME QUERY_NAME CMD_NAME"
```

where

- `ITEM_NAME` gives the name of the enumeration value in `OPT_Options.java`
- `QUERY_NAME` gives the name of an accessor function which returns `true` iff the enumeration takes the value `ITEM_NAME`.
- `CMD_NAME` is the name to pass on the command-line to set the enumeration to this value.

The quotes are important, and the specifications should be space-separated.

For example, RVM supports a choice of three options for floating-point optimization rules. The three-line record describing these options is:

```
E FP_MODE byte FP_STRICT
Selection of strictness level for floating point computations
"FP_STRICT strictFP strict" \
"FP_ALLOW_FMA allowFMA allow_fma" \
"FP_LOOSE allowAssocFP allow_assoc"
```

Notice how the third line was broken up by using "\".

So, by default, RVM uses the *strict* floating-point semantics. To use the option that allows fused multiply-add instructions, specify `-X:irc:allow_fma` on the command-line. Given an `OPT_Options` object called `options`, your code can query if fma is allowed by testing `options.allowFMA()`.

5.2 Method Compilation

The fundamental unit for optimization in RVM is a single method. The optimization of a method consists of a series of compiler phases performed on the method. These phases transform the IR (intermediate representation) from bytecodes through HIR (high-level intermediate representation), LIR (low-level intermediate representation), and MIR (machine intermediate representation) and finally into machine code. Various optimizing transformations are performed at each level of IR.

An object of the class `OPT_CompilationPlan` contains all the information necessary to generate machine code for a method. Two important component of this class are the `VM_Method` to be compiled and the array of `OPT_OptimizationPlanElements` to perform the compiling. When the `execute` method of this class is called, machine code is generated for the method as described by the `OPT_OptimizationPlanElement`s.

Another important class is `OPT_OptimizationPlanner`. This class contains a static field, called `masterPlan`, which contains all possible `OPT_OptimizationPlanElement`s. The structure of the master plan is a tree. Any element may either be an atomic element (a leaf of the tree), or an aggregate element (an internal node of the tree). The master plan has the following general structure:

- elements which convert bytecodes to HIR
- elements which perform optimization transformations on the HIR
  - elements which perform optimization transformations using SSA form
- elements which convert HIR to LIR
- elements which perform optimization transformations on the LIR
  - elements which perform optimization transformations using SSA form
- elements which convert LIR to MIR
- elements which perform optimization transformations on MIR
- elements which convert MIR to machine code

A specific optimization plan is constructed by including all the `OPT_OptimizationPlanElement`s contained in the master plan which are appropriate for this compilation instance. Whether or not an element should be part of a compilation plan is determined by its `shouldPerform` method. For each atomic element, the values in the `OPT_Options` object are generally used to determine whether the element should be included in the compilation plan. Each aggregate element must be included when any of its component elements must be included.

Each element must have a `perform` method defined which takes the IR as a parameter. It is expected, but not required, that the `perform` method will modify the IR. The perform method of an aggregate element will invoke the perform methods of its elements.

Each atomic element is an object of the final class `OPT_OptimizationPlanAtomicElement`. The main work of this class is performed by its *phase*, an object of type `OPT_CompilerPhase`. The `OPT_CompilerPhase` class is not final; each phase overrides this class, in particular it overrides the `perform` method, which is invoked by its enclosing element's `perform` method. All the state associated with the element is contained in the `OPT_CompilerPhase`; no state is in the element.

Every optimization plan consists of a selection of elements from the master plan; thus though two optimization plans will be associated with different methods they will share the same element objects. Clearly, it is not desirable that any state associated with a particular compilation phase should be shared between two different methods. In order to prevent this , the `perform` method of an atomic element creates a new instance of its phase immediately before calling the phase's `perform` method. In the case where the phase contains no state the `newExecution` method of `OPT_CompilerPhase` can be overridden to return the phase itself rather than a clone of the phase.

5.3 IR Operators

The optimizing compiler intermediate representation (IR) includes a list of instructions. Each instruction includes an operator and zero or more operands.

The IR operators are defined by the class `OPT_Operators`, which in turn is automatically generated from a template by a driver. The input to the driver are two files, both called `OperatorList.dat`. One input file resides in `$RVM_ROOT/rvm/src/vm/compilers/optimizing/ir/instruction` and defines machine-independent operators. The other resides in `$RVM_ROOT/rvm/src/vm/arch/{arch}/compilers/optimizing/ir/instruction` and defines machine-dependent operators, where {arch} is the specific architecture of interest, such as PowerPC™[12].

Each operator in `OperatorList.dat` is defined by a five-line record, consisting of:

- `SYMBOL`: a static symbol to identify the operator
- `INSTRUCTION_FORMAT`: the instruction format class that accepts this operator. See Section 5.4 for more information.
- `TRAITS`: a set of characteristics of the operator, composed with a bit-wise or (|) operator. See `OPT_Operator.java` for a list of valid traits.
- `IMPLDEFS`: set of registers implicitly defined by this operator; usually applies only to machine-dependent operators
- `IMPLUSES`: set of registers implicitly used by this operator; usually applies only to machine-dependent operators

For example, the entry in `OperatorList.dat` that defines the integer addition operator is

```
INT_ADD
Binary
none
<blank line>
<blank line>
```

The operator for a conditional branch based on values of two references is defined by

```
REF_IFCOMP
IntIfCmp
branch | conditional
<blank line>
<blank line>
```

Additionally, the machine-specific `OperatorList.dat` file contains another line of information for use by the assembler. See the file for details.

5.4 Instruction Formats

Every IR instruction fits one of the pre-defined *Instruction Formats*. The package `instructionFormats.java` defines roughly 75 architecture-independent instruction formats. For each instruction format, the package includes a class that defines a set of static methods by which optimizing compiler code can access an instruction of that format.

For example, `INT_MOVE` instructions conform to the `Move` instruction format. The following code fragment shows code that uses the `OPT_Operators` interface and the `Move` instruction format:

[12] **PowerPC** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

```
import instructionFormats.*;
class X {
  void foo(OPT_Instruction s) {
    if (Move.conforms(s)) {     // if this instruction fits the Move format
      OPT_RegisterOperand r1 = Move.getResult(s);
      OPT_Operand r2 = Move.getVal(s);
      System.out.println("Found a move instruction: " + r1 + " := " + r2);
    } else {
      System.out.println(s + " is not a MOVE");
    }
  }
}
```

This example shows just a subset of the access functions defined for the Move format. Other static access functions can set each operand (in this case, `Result` and `Val`), query each operand for nullness, clear operands, create Move instructions, mutate other instructions into Move instructions, and check the index of a particular operand field in the instruction. See the javadoc reference for a complete description of the API.

Each fixed-length instruction format is defined in the text file `$RVM_ROOT/rvm/src/vm/compilers/optimizing/ir/instruction/InstructionFormatList.dat`. Each record in this file has four lines:

- `NAME`: the name of the instruction format
- `SIZES`: the number of operands defined, defined and used, and used
- `SIG`: a description of each operand, each description given by
  - `D/DU/U`: Is this operand a def, use, or both?
  - `NAME`: the unique name to identify the operand
  - `TYPE`: the type of the operand (a subclass of `OPT_Operand`
  - `[opt]`: is this operand optional?
- `VARSIG`: a description of repeating operands, used for variable-length instructions.

So for example, the record that defines the `Move` instruction format is

```
Move
1 0 1
"D Result OPT_RegisterOperand" "U Val OPT_Operand"
<blank line>
```

This specifies that the `Move` format has two operands, one def and one use. The def is called `Result` and must be of type `OPT_RegisterOperand`. The use is called `Val` and must be of type `OPT_Operand`.

A few instruction formats have variable number of operands. The format for these records is given at the top of `InstructionFormatList.dat`. For example, the record for the variable-length `Call` instruction format is:

```
Call
1 0 3 1 U 4
"D Result OPT_RegisterOperand" \
"U Address OPT_Operand" "U Method OPT_MethodOperand" "U Guard OPT_Operand opt"
"Param OPT_Operand"
```

This record defines the `Call` instruction format. The second line indicates that this format always has at least 4 operands (1 def and 3 uses), plus a variable number of uses of one other type. The trailing 4 on line 2 tells the template generator to generate special constructors for cases of having 1, 2, 3, or 4 of the extra operands. Finally, the record names the `Call` instruction operands and constrains the types. The final line specifies the name and types of the variable-numbered operands. In this case, a `Call` instruction has a variable number of (use) operands called `Param`. Client code can access the `i`th parameter operand of a `Call` instruction `s` by calling `Call.getParam(s,i)`.

A number of instruction formats share operands of the same semantic meaning and name. For convenience in accessing like instruction formats, the template generator supports four common operand access types:

- `ResultCarrier`: provides access to an operand of type `OPT_RegisterOperand` named `Result`.
- `GuardResultCarrier`: provides access to an operand of type `OPT_RegisterOperand` named `GuardResult`.
- `LocationCarrier`: provides access to an operand of type `OPT_LocationOperand` named `Location`.
- `GuardCarrier`: provides access to an operand of type `OPT_Operand` named `Guard`.

For example, for any instruction `s` that carries a `Result` operand (eg. `Move`, `Binary`, and `Unary` formats), client code can call `ResultCarrier.conforms(s)` and `ResultCarrier.getResult(s)` to access the `Result` operand.

Finally, a note on rationale. Religious object-oriented philosophers will cringe at the InstructionFormats. Instead, all this functionality could be implemented more cleanly with a hierarchy of instruction types exploiting (multiple) inheritance. We rejected the class hierarchy approach due to efficiency concerns of frequent virtual/interface method dispatch and type checks. Recent improvements in our interface invocation sequence and dynamic type checking algorithms may alleviate some of this concern.

5.5 BURS Rules

The optimizing compiler uses the Bottom-Up Rewrite System (BURS) for instruction selection. BURS is essentially a tree pattern matching system derived from Iburg by David R. Hanson. (See "Engineering a Simple, Efficient Code-Generator Generator" by Fraser, Hanson, and Proebsting, LOPLAS 1(3), Sept. 1992.) The instruction selection rules for each architecture are specified in an architecture-specific file called `LIR2MIR.rules`, which resides in `$RVM_ROOT/rvm/src/vm/arch/{arch}/compilers/optimizing/ir/conversions/lir2mir`, where {arch} is the specific architecture of interest, such as PowerPC™[13]. The rules are used in generating a parser, which transforms the IR.

Each rule in `LIR2MIR.rules` is defined by a four-line record, consisting of:

- `PRODUCTION`: the tree pattern to be matched. The format of each pattern is explained below.
- `COST`: the cost of matching the pattern as opposed to skipping it. It is a Java expression that evaluates to an integer.
- `FLAGS`: specifies whether the rule actually represents a sequence of instructions (EMIT_INSTRUCTION) or a transformation of operands (NOFLAGS). Other flags can be used to control the order of code generation of a node's children.

[13] **PowerPC** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

- TEMPLATE: Java code to emit

Each production has a *non-terminal*, which denotes a value, followed by a colon (":"), followed by a dependence tree that produces that value. For example, the rule resulting in memory add on the INTEL architecture is expressed in the following way:

```
stm:    INT_STORE(INT_ADD_ACC(INT_LOAD(r,riv),riv),OTHER_OPERAND(r, riv))
ADDRESS_EQUAL(P(p), PLL(p), 17)
EMIT_INSTRUCTION
EMIT(MIR_BinaryAcc.mutate(P(p), IA32_ADD, MO_S(P(p), DW), \
                          BinaryAcc.getValue(PL(p))));
```

The production in this rule represents the following tree:

```
         r     riv
          \    /
         INT_LOAD  riv
            \     /
           INT_ADD_ACC  r  riv
                   \   |  /
                  INT_STORE
```

where r is a non-terminal that represents a register or a tree producing a register, riv is a non-terminal that represents a register (or a tree producing one) or an immediate value, and INT_LOAD, INT_ADD_ACC and INT_STORE are operators (*terminals*). OTHER_OPERAND is just an abstraction to make the tree binary.

There are multiple helper functions that can be used in Java code (both cost expressions and generation templates). In all code sequences the name p is reserved for the current tree node. Some of the helper methods are shortcuts for accessing properties of tree nodes:

- P(p) is used to access the instruction associated with the current (root) node,
- PL(p) is used to access the instruction associated with the left child of the current (root) node (provided it exists),
- PR(p) is used to access the instruction associated with the right child of the current (root) node (provided it exists),
- similarly, PLL(p), PLR(p), PRL(p) and PRR(p) are used to access the instruction associated with the left child of the left child, right child of the left child, left child of the right child and right child of the right child, respectively, of the current (root) node (provided they exist).

What the above rule basically reads is the following:
If a tree shown above is seen, evaluate the cost expression (which, in this case, calls a helper function to test whether the addresses in the STORE (P(p)) and the LOAD (PLL(p)) instructions are equal. The function returns 17 if they are, and a special value INFINITE if not), and if the cost is acceptable, emit the STORE instruction (P(p)) mutated in place into a machine-dependent add-accumulate instruction (IA32_ADD) that adds a given value to the contents of a given memory location.

The rules file is used to generate a file called ir.brg, which, in turn, is used to produce a file called OPT_BURS_STATE.java.

For more information on helper functions look at $RVM_ROOT/rvm/src/vm/arch/{arch}/compilers/optimizing/ir/conversions/lir2mir/OPT_BURS_Helpers.java. For more information on the BURS algorithm see $RVM_ROOT/rvm/src/vm/compilers/optimizing/ir/conversions/lir2mir/OPT_BURS.java.

5.6 "Magic" Methods

Certain methods, known as "magic" methods and declared as static methods of `VM_Magic`, are treated differently by the compiler. Because these methods access raw memory (such as portions of object headers) or registers, or are operating system calls they cannot be implemented in Java code. Instead, for each of these methods, the Java instructions to generate the code is stored in `OPT_GenerateMagic` and `OPT_GenerateMachineSpecificMagic` (to generate HIR) and `VM_MagicCompiler` (to generate assembly code)[14].

When a call site is being compiled, and it is determined that the call target is one of these magic methods, control is transferred to the list of instructions which will generate the HIR for that method. The HIR for the magic method is always inlined into the caller method.

As an RVM implementor, you must be *extremely careful* when writing code that uses `VM_Magic` to circumvent the Java type system. The use of `VM_Magic.objectAsAddress` to perform various forms of pointer arithmetic is especially hazardous, since it can result in pointers being "lost" during garbage collection. All such uses of magic must either occur in uninterruptible code (ie in a method of a class that *directly* implements `VM_Uninterruptible`) or be guarded by calls to `VM.disableGC` and `VM.enableGC`. The optimizing compiler performs aggressive inlining and code motion and not explictly marking such dangerous regions in one of these two manners will lead to disaster.

5.7 Adaptive Optimization System

For a comprehensive discussion of the design and implementation of the adaptive optimization system, see our 2000 OOPSLA paper.

For details on individual classes, see the source files under `$RVM_ROOT/rvm/src/vm/adaptive`.

In summary, version 2.0 of the Jikes RVM includes basic AOS functionality to identify and recompile program hot spots, context-insensitive online profile-directed inlining, and basic support for dynamic instrumentation.

5.8 OptTestHarness

For optimizing compiler development, it is sometimes useful to exercise careful control over which classes are compiled, and with which optimization level. In many cases, a `BaseOptSemispace` image will suit this process. This configuration invokes the optimizing compiler on each method run. Since the optimizing compiler is not in the boot image, you can modify its classes without relinking the boot image. Instead, `jbuild -nolink` will recompile the source files you edit, without invoking the time-consuming boot image writing step.

The `OptTestHarness` program provides even more control over the optimizing compiler. This driver program allows you to invoke the optimizing compiler as an "application" running on top of the VM. The most useful configuration for this is probably `BaseBaseOTHcopyingGC`; this image is a BaseBase image which includes just enough support to invoke the optimizing compiler through `OptTestHarness`. Like the `BaseOpt` images, you can use `jbuild -nolink` to skip a time-consuming boot image writing during development.

To use the `OptTestHarness` program:

```
% rvm OptTestHarness -class Foo
```

will invoke the optimizing compiler on all methods of class `Foo.`

```
% rvm OptTestHarness -method Foo bar -
```

will invoke the optimizing compiler on the first method `bar` of class `Foo` it loads.

[14]The optimizing compiler always uses the set of instructions that generate HIR; the instructions that generate assembly code are only invoked by the baseline compiler.

```
% rvm OptTestHarness -method Foo bar (I)V;
```

will invoke the optimizing compiler on method `Foo.bar(I)V;`.

You can specify any number of `-method` and `-class` options on the command line. Any arguments passed to `OptTestHarness` via `-oc` will be passed on directly to the optimizing compiler. So:

```
% rvm OptTestHarness -oc:O1 -oc:print_final_hir=true -method Foo bar -
```

will invoke compile `Foo.bar` at optimization level `O1` and print the final HIR.

One other useful option to `OptTestHarness` is `-longcommandline <filename>`. With this option, `OptTestHarness` reads the command line from a file.

The source to the `OptTestHarness` resides in `$RVM_ROOT/rvm/src/tools/optTestHarness`.

6 Memory Management Details

This Section provides additional information on the implementation of the memory management component of RVM runtime system.

6.1 Directory Structure

The classes related to memory management are contained in the `$RVM_ROOT/rvm/src/vm/memoryManagers` directory. The `watson` sub-directory of `memoryManagers` contains the memory managers provided with the initial release of the RVM, and are the managers discussed in this section.

The `watson` directory contains common classes used by multiple memory managers, such as the load balancing work queue (see `VM_GCWorkQueue.java`). Sub-directories of `watson` contain classes for a specific memory management strategy. For example, the `semispace` directory implements a simple semi-space copying allocator and collector.

A RVM configuration specifies a memory management strategy, and the jconfigure command will include in the build all the classes in the `memoryManagers/watson` directory and all the classes in the sub-directory for the specified memory management strategy.

6.2 Choosing an Allocator/Collector

Depending on your purposes, you may choose to build RVM with a semispace copying, mark-sweep noncopying, or hybrid collector. For programs which do not perform many collections, the copying collector will most likely perform best, since object allocation path lengths are shortest for this collector. To minimize garbage collection delays, the hybrid collector may be used, since it allows a small nursery for which minor collections can be quite fast, and major collections are noncopying.

Generally, the mark-sweep noncopying collector requires smaller heap storage for a given application than any of the copying collectors, which in have a semispace structure. Also, major collections are faster with a noncopying collector, since objects are not moved. The best performance for a particular application should be determined by experiment.

All of the memory managers provided in the `watson` directory, except for the concurrent (reference counting) collector, support finalization, and collection while threads are executing in native code (such threads are blocked in native code during collection).

6.3 Adding A New Allocator/Collector

It is not difficult to add your own Allocator/Collector to RVM, especially if it uses the same "stop the world" parallel collection strategy used by all the collectors in this release. The basic steps are:

1. Create a new directory for the Allocator/Collector, such as "newGC".
2. Add a new configuration in `$RVM_ROOT/rvm/config/build` which includes your new directory in the build. Name it appropriately, such as "BaseBasenewGC".
3. Copy some existing `VM_Allocator.java` file into your new directory, and modify it, choosing one that has similar properties, such as copying or non-copying. If you want to start from scratch, start with `VM_Allocator` in noGC. This file provides simple implementations of the required object allocation methods, and stubs for the other fields and methods expected by the rest of the RVM runtime (not all will be needed in any one implementation). With the initial set of memory managers, a single class, VM_Allocator, implements both the methods that perform object allocation and the methods that perform garbage collections.

6.4 Load Balancing Work Queue

The class `VM_GCWorkQueue`, implements a load balancing workqueue which is used by all the collectors to find all references reachable from some initial set of references. It is used to find all objects reachable from roots, and also to find all objects reachable from finalizable objects that are made live in order to execute their finalizer methods.

The work queue is implemented using thread local "get" and "put" buffers and a shared pool of buffers needing to be processed. The performance of the work queue is affected by the size of these buffers. The default size is 1024 entries, but can be altered by the command line argument "-X:wbsize=nnn" where nnn is the maximum number of entries in the buffer. When running with multiple processors, better load balancing has been observed with smaller buffers, for example with 256 entries per buffer.

6.5 Generational Write Barrier

RVM provides a class `VM_WriteBarrier` to support generational garbage collection. When the static final field `VM_Allocator.writeBarrier` is true, the RVM compilers generate write barriers. The baseline compiler generates calls to barrier methods in `VM_WriteBarrier`. The optimizing compiler inlines the same barrier methods.

The logic of the write barrier is implemented in `VM_WriteBarrier`. The RVM generational collectors all use the same write barrier. It stores into a processor local write buffer the address of an object into which a ref is being stored. A bit in the object header identifies objects that need to be recorded. The barrier logic turns off the bit when an object is written to the write buffer, to avoid duplicate entries. During garbage collection, collectors must set these bits back on so that subsequent stores into the object will cause it to be recorded.

Alternative write barriers can be implemented. For example, the concurrent reference counting collector uses a barrier that records both old and new references for each store of a reference into an object. It does this by building with its own version of VM_WriteBarrier.

The RVM generational collectors treat all objects which survive one garbage collection as old, and do not maintain "remembered sets" of old objects between collections.

6.6 Starting Garbage Collection

The stopping of mutator threads and the scheduling of the collector threads is handled by the classes `VM_Handshake` and `VM_CollectorThread`. Mutator threads initiate collection by calling the `collect` method of `VM_CollectorThread` which causes collector threads to be scheduled on all the `VM_Processor`s, and then yields, to allow the collector thread on the executing processor to be scheduled. Parallel execution of collector threads is synchronized by rendezvous. During the initial rendezvous, one of the collector threads detects processors whose executing thread is blocked in native code, and makes these processors "non-participating" for that collection. Collection begins when all "participating" collector threads arrive at the initial rendezvous. After the initial rendezvous, the participating collector threads call the `collect` method of `VM_Allocator`. They execute in parallel in this method until collection is complete. By default, there is one collector thread for each `VM_Processor` (system thread). While collection is in progress, all non-collector threads are suspended in queues of the `VM_Processors` waiting to be re-dispatched.

To implement other than stop-the-world parallel collection (such as was done with the concurrent collector) then it will be necessary to modify or extend the `VM_Handshake` and `VM_CollectorThread` classes.

6.7 Measuring Collector Performance

When the "-verbose:gc" command line argument is specified, the time spent in each collection will be written out after each collection. In addition, summary statistics are generated when RVM exits, which specify the number of collections, and average and maximum collection times. For generational collectors, this is given for Minor and Major collections. Some collectors provide additional information, that can be measured with minimal cost, such a number of bytes copied for the copying collectors.

Compile time flags can be set to cause additional information, at additional cost, to be measured and reported. Some of the more useful ones are described here, others are described in the various `VM_Allocator` source files.

TIME_GC_PHASES This flag will cause the collector to measure the time spent in each of the phases of garbage collection, such as stopping mutators, finding roots, marking reachable objects, and finalization. Turning on this flag will cause the summary statistics, with average times in each phase, to be generated when RVM exits. If `-verbose:gc` is specified the output is generated after each collection. Turning on `TIME_GC_PHASES` in the common `VM_CollectorThread` class will cause it to be set in what ever `VM_Allocator` is being used.

MEASURE_WAIT_TIMES This flag will cause the collector to measure the time each collector thread spends waiting during a collection. This is time waiting for buffers while processing the Work Queue, and time waiting in rendezvous between phases of the collection process. Turning on this flag will cause the summary statistics, with average wait times, to be generated when RVM exits. If -verbose:gc is specified the output is generated after each collection, for each collector thread. Turning on `MEASURE_WAIT_TIMES` in the common `VM_CollectorThread` class will cause necessary flags in `VM_Allocator` and `VM_GCWorkQueue` to be set.

RENDEZVOUS_TIMES prints per thread entrance and exit times for the rendezvous during each collection.

GC_COUNT_BY_TYPES prints counts of the number of live objects grouped by type.

WORK_QUEUE_COUNTS counts work queue buffers processed by each collector thread (in VM_GCWorkQueue).

COUNT_GETS_AND_PUTS counts object references processed by each collector thread (in VM_GCWorkQueue).

Not all flags are available in all VM_Allocator files.

7 JDP: the Jikes RVM debugger

This section provides information regarding `jdp`, the RVM debugger.

7.1 About `jdp`

`jdp`, the RVM Debugger Primitive, is a low-level symbolic debugger developed to support the RVM effort. Because no existing debugger satisfactorily meets the needs of our RVM, `jdp` was written essentially from scratch, hence the name primitive. `jdp` uses a technique called Remote Reflection (see the related paper on the RVM web page). This allows `jdp` to run out-of-process, yet it uses reflection extensively to access the internal data structures as if it resides inside the RVM.

`jdp` is written in the Java™[15] programming language and JNI with native code in C. `jdp` uses the AIX™[16] `ptrace` interface and the standard Unix process support (fork, wait, ...). When porting to another platform, this native portion (called from Platform.java) will have to be rewritten. Since `jdp` and the RVM being debugged are two separate processes, an interpreter is used to connect `jdp` with the RVM: it maps the access bytecodes in the reflection methods to obtain the internal values in the RVM.

`jdp` is not intended to be a general purpose debugger; it is customized for RVM but is easily extendable.

Because the optimizing compiler does not currently provide all of the mapping support required, `jdp` functionality is limited in opt-compiled code.

7.2 Getting started

You can use `jdp` in one of three modes: debugging the boot image only, general debugging, and debugging a RVM that is already running:

1. Use the boot image only mode if you are only debugging code in the boot image since this is faster and the debugger is more stable (even when the RVM becomes corrupted):

   ```
   jdp -jdpbootonly myProgram myArguments
   ```

   In this mode, methods of classes outside the boot image appear as "unknown method" and you cannot set breakpoint for these methods.

2. Use the general mode if you are debugging codes in classes that are dynamically loaded:

   ```
   jdp myProgram myArguments
   ```

   This mode is slower because there are two levels of interpretation and the debugger can get lost if the dictionaries in the RVM become corrupted.

3. If the RVM is already running and appears to be hung, you can attach `jdp` to this RVM by:

   ```
   jdp -jdpattach<processID>
   ```

   The process ID is obtained from the "ps" command. The option "`-jdpbootonly`" can be used in conjunction with "`-jdpattach`". The effect is that the debugger is faster and more stable, but it will only show methods in the boot image.

[15] **Java** and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

[16] **AIX** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

4. If there is an error in the RVM initialization, you can start debugging at a point where the boot image has been loaded but the RVM has not begun executing yet. Specify the option:

   `jdp -jdpviewboot`

   Note that if the option "`-jdpbootonly`" is used, the option "`-jdpviewboot`" is also on by default because jdp would be unable to find the dynamically loaded user's main method.

7.3 `jdp` environment

This section describes the files and settings that the debugger needs. The `jdp` command is a `ksh` script that does some preliminary argument parsing and fills in default arguments before invoking the debugger in the appropriate mode.

The debugger requires the following files:

1. The RVM boot image: This file is typically `$RVM_BUILD/RVM.image` but it could have other names with the `.image extension`. You can specify a different boot image name.
   - `jdp` argument: -i bootImageName
   - Default value: `$RVM_BUILD/RVM.image`
2. The symbol map: This file is a symbol map of the boot image. It contains a listing of the JTOC and the compiled methods. It is generated when the build configuration in `$RVM_ROOT/rvm/bin/jconfigure` contains the flag: `export GENERATE_MAP=1`

   `jdp` uses offset values from this file to find all symbols in the boot image. In the dynamic mode, the interpreter also uses this file to find the base offset for the dictionaries for the remote reflection feature.
   - `jdp` argument: (none)
   - Default value: `$RVM_BUILD/RVM.map`
3. The list of classes in the boot image: `jdp` looks for the file `$RVM_BUILD/RVM.primordials`
   - `jdp` argument: `-n classListName`
   - Default value: `$RVM_BUILD/RVM.primordials`
4. The program to load and run the RVM boot image: This file is typically `$RVM_BUILD/JikesRVM`. Normally you don't have to be concerned about this file since it rarely changes.
   - `jdp` argument: `-jdpbootrunner booterName`
   - Default value: `$RVM_BUILD/JikesRVM`

In addition to these files, the following setting can be made:

1. Initial breakpoint: This is where the RVM will stop first. By default, `jdp` stops after the prologue of the user's main method. However, this point comes after the RVM has executed for a significant amount of time. If there is an error in the initialization of the RVM, `jdp` can stop before the RVM begins execution by specifying the argument `-jdpviewboot`. In this case, `jdp` sets the initial breakpoint at a point after the first 4 instructions in the assembler procedure .bootThread in: `$RVM_ROOT/rvm/src/tools/bootImageRunner/bootThread.c`. This is after the RVM boot image has been loaded into memory and the 4 registers jtoc, proc, thread index and FP have been initialized. At this point, only one thread is running and the stack has only one frame pointing to the booting C code.

- `jdp` argument: `-jdpbreakpoint breakpointHexValue`
- Default value: bootThread

2. Process ID: This is used for attaching `jdp` to a currently running RVM. After initializing itself, `jdp` will send an interrupt to the running process to gain control. The process ID is from the "`ps`" command in AIX$^{\mathrm{TM}}$[17].

   - `jdp` argument: `-jdpattachXXXX` where XXXX is the process ID
   - Default value: (none)

Finally, jdp assumes the following conventions in navigating within the boot image (they arise from the base compiler convention):

1. The method ID is saved a the end of the instruction block for each method.
2. The end of the prolog of the instruction block is marked by a code pattern. For AIX/PowerPC$^{\mathrm{TM}}$[18], it is:

```
4ffffb82
```

   which is the instruction:

```
cror   0x1f,0x1f,0x1f
```

   For Linux/Intel, it is:

```
90
```

   which is the NOP instruction:

[17]**AIX** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

[18]**PowerPC** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

7.4 jdp commands

Command	Shortcut	Description
step	s	step current thread by instruction, into method
stepbr	sbr	step current thread by instruction, over method
stepline	sl	step current thread by source line, into method
steplineover	slo	step current thread by source line, over method
creturn	cr	continue to calling method (up one stack frame)
cont	c	continue all threads
kill	k	terminate program
run	run	start new program
break	b	list/set breakpoint
clearbreak	cb	clear breakpoints
thread	th	select or turn off thread context
where	w	print short stack trace
whereframe	wf	print full stack trace
stack	f	display formatted stack
mem	m	display memory
memraw	mraw	display actual memory (jdp breakpoints are visible)
wmem	wm	write memory
reg	r	display registers
wreg	wr	write register
printclass	pc	print the class statics or the type of an object address
ymacroprint	p	print local variables or cast an address as an object
listi	li	list machine instruction
listt	lt	list threads
quit	q	exit debugger
preference	pref	set user preference
x2d, d2x	(same)	convert number between hex and decimal
verbose	v	toggle verbose mode
(macro name)		load and execute this macro (a text file with suffix .jdp)
(enter)		repeat last command

To get more information on a specific command, type:

```
help thiscommand
```

7.5 jdp Macros

A jdp macro is a text file that has the `.jdp` suffix and contains a sequence of normal `jdp` commands

- When you type `m` at the command line, `jdp` will look for `mymacro.jdp` in the normal class path and execute each line in the macro file as if they are entered at the command line.
- On entry to `jdp`, if the file `startup.jdp` exists in the current directory, it will be loaded and executed automatically.

This should be convenient for:

- Short cut to repeat a series of commands to get a debugging point
- Regression testing: redirect the output and compare with previous results. This is specifically intended for regression testing of `jdp` itself but may be useful for other tests as well, especially if the test needs to inspect specific runtime values in registers, memory.

7.6 Debugging tips

- Compiling the system with local variables: The default boot image is built without the local variable tables. To get local variables, set this flag to true in your local shadow:

  ```
  VM_Control.LoadLocalVariableTables
  ```

  Then rebuild the boot image. This will increase the boot image size by about 10%.

- Setting breakpoint: Sometimes a breakpoint cannot be set. In the boot image only mode, `jdp` cannot find code outside the boot image. In the general mode, `jdp` cannot find codes for classes that have not been loaded and compiled, simply because they don't exist yet. To get around these situations, the boot image contains a dummy method which `jdp` can always find: `VM.debugBreakpoint`. You can insert a call to this method in the code where you want to stop. Then set a breakpoint on this method, proceed to this breakpoint, then continue to the caller to reach your method:

  ```
  b VM.debugBreakpoint
  c
  cr
  ```

  In the general debugging mode, when `jdp` stops at this breakpoint, the class containing the calling method has been loaded and you can set then breakpoints in any method of this class.

- Setting breakpoint in the prolog: Normally, `jdp` skips past the prolog code when it sets a breakpoint in a method. To force `jdp` to set a breakpoint at the beginning of the method prolog, specify 0 as the line number:

  ```
  b myclass.mymethod:0
  ```

- Expanding expression: Symbolic expression for printing class static variable and stack local variable can be nested arbitrarily

  ```
  pc VM_Scheduler.threads[1].contextRegisters
  p 0:myLocalVar.field1.field2
  ```

- Casting an address as an object: If you have the address of an object and you know the class name, you can cast the address to print the object:

  ```
  p (className) xxxxxxxx
  ```

  This can also be used to print a concrete object of an abstract class. If the casting is not correct, you will see messages such as:

  ```
  CAUTION, address not accessible: 0x80620420
  ```

- Getting the class name of an object address: If you have the address of an object and want to know the type:

  ```
  pc xxxxxxxx
  ```

- Getting to low level string: Strings are stored as `VM_Atom`s in the RVM. To print the string as character, print the memory at the address of the field `VM_Atom.val`:

```
jdp:0>p (VM_Atom) 30266638
VM_Atom =
  VM_Atom @0x30266638
    val = {100, 101, 98, 117, 103, 66, 114, 101, 97, 107, 112, 111, 105, 110, 116} @0x30269628
    hash = 1544209252  @0x30266624

jdp:0>m 30269628
  0x30269628: 64 65 62 75    d e b u
  0x3026962c: 67 42 72 65    g B r e
  0x30269630: 61 6b 70 6f    a k p o
  0x30269634: 69 6e 74 00    i n t .
  0x30269638: 30 05 67 90    0 . g .
```

- Restarting your program: If your program runs to completion and you want to re-execute it, type "run" in `jdp`. A new RVM will be loaded and started in a new process. The existing breakpoints will be saved and set again when the boot image has been loaded.

- Display number in hex or decimal: Integers are normally displayed in decimal for variables and in hex for stack. Sometimes variables contain addresses as integer and stack entries contain small integer values. In these cases, it may be more convenient to view in the alternate format. To change the display format for variables to hex:

  ```
  pref int x
  ```

  To change the display format for stack to decimal:

  ```
  pref stack d
  ```

  Likewise, floating point registers are displayed as float, but to compare the actual values without any rounding, it may be more convenient to view the hex values. To change the format to hex:

  ```
  pref fpr x
  ```

  The format preference can be changed back by specifying x, d or f as appropriate.

- Interrupting a hung RVM: If the RVM is running under the debugger and appears to be hung, you can interrupt it by going to a different window and killing the RVM process. This will send a signal to the RVM process and `jdp` will catch the signal before it gets to the RVM process. Hitting control-c in the jdp window will kill `jdp` itself.

- Debugging with native code: `jdp` provides limited debugging for native codes that are interspersed via JNI. `jdp` displays the C procedure names in the stack traceback and stepping by machine code is possible. You can also view the raw C stack frame and id its registers, but native variables and line numbers are not available. For C program that creates the RVM via the JNI_CreateJavaVM call, `jdp` can be used by specifying the initial breakpoint (see the `jdp` script).

- Debugger error messages: What these messages mean:

  - `"CAUTION, address not accessible:  0x80620420"` This means that `jdp` got an error when it reads an address in the RVM space. This may have occurred because `jdp` is dereferencing an object from a bad address, or the RVM has become corrupted. This message does not mean that `jdp` has crashed; you should be able to continue debugging.

– `j`: In the general debugging mode (running with the interpreter), this means that the interpreter has encountered an internal error and has entered its own internal debugging mode. At this point, you can type "w" to see the interpreter stack and report the problem.

8 Using the Jikes RVM to Profile an Application

This section contains information on how an adaptive configuration of RVM can be used to profile an application and the VM. The first mechanism provides a coarse-grain profile, giving the percentage of execution time spent in the hottest methods. The second method provides a mechanism to insert counters to count the frequency of specific events.

8.1 Profiling An Application

One component of the adaptive optimization system is a low-overhead time-based sampling mechanism. This information can be used to drive recompilation decisions [2]. It can also be used to produce an aggregate profile of the execution of an application. Here's how.

1. Create an adaptive configuration. For the most accurate profile use `FastAdaptiveSemispace`. See Section 2

   ```
   % jconfigure FastAdaptiveSemispace
   % cd $RVM_BUILD
   % jbuild
   ```

2. Run the application using the opt compiler as the runtime compiler and instructing RVM to gather profile data.

   ```
   % rvm -X:aos:primary_strategy=optonly -X:aos:gather_profile_data=true <classfile>
   ```

8.2 Instrumented Event Counters

This section describes how the RVM optimizing compiler can be used to insert counters in the optimized code to count the frequency of specific events. Infrastructure for counting events is in place that hides many of the implementation details of the counters, so that (hopefully) adding new code to count events should be easy. All of the instrumentation phases described below require an adaptive boot image (any one should work). Most of the code regarding instrumentation lives in `$RVM_ROOT/rvm/src/vm/adaptive/runtimeMeasurements/instrumentation` and `adaptive/recompilation/instrumentation`.

Section 8.2.1 describes existing instrumentation phases and how to run them, and Section 8.2.2 describes the details of how a new phase can be added.

8.2.1 Existing instrumentation phases

There several existing instrumentation phases. For now, turning on each phase requires setting two flags, one for the AOS system, and one for the opt compiler. These counters are *not* synchronized (as discussed in Section 8.2.2), so they should not be considered precise.

1. **Method Invocation Counters**

   Inserts a counter in each method prologue. Prints counters to stderr at end.

   Parameters:
   `-X:aos:primary_strategy=optonly`
   `-X:aos:share:insert_method_counters_opt=true`

2. **Yieldpoint Counters**

   Inserts a counter after each yieldpoint instruction. Maintains a separate counter for backedge and prologue yieldpoints.

Parameters:
`-X:aos:primary_strategy=optonly`
`-X:aos:share:insert_yieldpoint_counts=true`

3. **Instruction Counters**

   Inserts a counters on each instruction. A separate count is maintained for each opcode, and results are dumped to stderr at end of run. The results look something like:

```
Printing Instruction Counters:
------------------------------
109.0 call
0.0 int_ifcmp
30415.0 getfield
20039.0 getstatic
63.0 putfield
20013.0 putstatic
Total: 302933
```

   This is useful for debugging or assessing the effectiveness of an optimization because you can see a dynamic execution count, rather than relying on timing.

   NOTE: Currently the counters are inserted at the end of HIR, so the counts *will* capture the effect of HIR optimizations, and will *not* capture optimization that occurs in LIR or later.

4. **Debugging Counters**

   This flag does not produce observable behavior by itself, but is designed to allow debugging counters to be inserted easily in opt-compiler to help debugging of opt-compiler transformations. If you would like to know the dynamic frequency of a particular event, simply turn on this flag, and you can easily count dynamic frequencies of events by calling the method `VM_AOSDatabase.debuggingCounterData.getCounterInstructionForEvent(String eventName);`. This method returns an `OPT_Instruction` that can be inserted into the code. The instruction will increment a counter associated with the String name "eventName", and the counter will be printed at the end of execution.

   For an example, see `OPT_Inliner.java`. Look for the code guarded by the flag `COUNT_FAILED_METHOD_GUARDS`.

   Parameters:
   `-X:aos:primary_strategy=optonly`
   `-X:aos:share:insert_debugging_counters=true`

8.2.2 Writing new instrumentation phases

This subsection describes the event counting infrastructure. It is not a step-by-step for writing new phases, but instead is a description of the main ideas of the counter infrastructure. This description, in combination with the above examples, should be enough to allow new users to write new instrumentation phases.

Counter Managers: Counters are created and inserted into the code using the `OPT_InstrumentedEventCounterManager` interface. The purpose of the counter manager interface is to abstract away the implementation details of the counters, making instrumentation phases simpler and allowing the counter implementation to be changed easily (new counter managers can be used without changing any of the instrumentation phases). Currently there exists only one counter manager, `VM_CounterArrayManager` , which implements unsynchronized counters. When instrumentation options are turned on in the adaptive system, `VM_Instrumentation.boot()` creates an instance of a `VM_CounterArrayManager`.

Managed Data: The class `VM_ManagedCounterData` is used to keep track of counter data that is managed using a counter manager. This purpose of the data object is to maintain the mapping between the counters themselves (which are indexed by number) and the events that they represent. For example, `VM_StringEventCounterData` is used record the fact that counter #1 maps to the event named "FooBar".

Depending on what you are counting, there may be one data object for the whole program (such as `VM_YieldpointCounterData` and `VM_MethodInvocationCounterData`), or one per method. There is also a generic data object called `VM_StringEventCounterData` that allows events to be give string names (see Debugging Counters above).

Instrumentation Phases: The instrumentation itself is inserted by a compiler phase. (see `OPT_InsertInstructionCounters.java`, `OPT_InsertYieldpointCounters.java`, `OPT_InsertMethodInvocationCounter.java`). The instrumentation phase inserts high level "count event" instructions (which are obtained by asking the counter manager) into the code. It also updates the instrumented counter to remember which counters correspond to which events.

Lower Instrumentation Phase: This phase converts the high level "count event" instruction into the actual counter code by using the counter manager. It currently occurs at the end of LIR, so instrumentation can not be inserted using this mechanism after LIR. This phase does not need to be modified if you add a new phase, except that the `shouldPerform()` method needs to have your instrumentation listed, so this phase is run when your instrumentation is turned on.

9 Experimental Guidelines

This section provides some tips on collecting performance numbers with RVM.

9.1 Which boot image should I use?

To make a long story short, use the

- `FastSemispace` configuration for performance runs that invoke the optimizing compiler on every method, and
- `FastAdaptiveSemispace` configuration for performance runs that use the adaptive compilation system.

These two configurations share the following characteristics:

- The code placed in the boot image is optimized.
- The optimizing compiler and associated support are in the boot image. Other configurations with this characteristic begin with the prefix `Full`. If you do not use a `Fast<...>` or `Full<...>` configuration, the optimizing compiler loads at runtime, and the optimizing compiler itself will be baseline compiled and run slowly.
- Both configurations set the final static boolean `VM.VerifyAssertions = false`. This flag avoids expensive assertion checking at runtime.
- Both configurations use the non-generational, copying (semi-space) collector. This collector has the fastest allocation sequence. Naturally, GC research will need to build configurations with other collectors.

9.2 What commmand-line arguments should I use?

For best performance we recommend the following:

- `-processors all`: By default, RVM uses only one processor. Setting this option tells the runtime system to utilize all available processors.
- `-X:irc:O2`: For non-adaptive configurations, this command-line option tells the optimizing compiler to use our highest level of optimization.
- Set the heap and large heap sizes generously. We typically set the heap size to at least half the physical memory on a machine.
- Use a dedicated machine with no other users. The RVM thread and synchronization implementation do not play well with others.

9.3 RVM is really slow! What am I doing wrong?

Perhaps you are not seeing stellar RVM performance. If RVM as described above is not competitive with the IBM AIX^{TM}[19]product DK, we recommend you test your installation with the SPECjvm98 benchmarks. We expect RVM performance to be competitive with the IBM AIX product DK 1.3.0 on the SPECjvm98 benchmarks.

[19] **AIX** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

Of course, SPECjvm98 does not guarantee that RVM runs all codes well. We have also tested various flavors of pBOB and the Volano benchmarks, and usually see superior or competitive performance.

The IA32 port is less mature than the PPC port, and does not deliver competitive performance on many codes. In particular, IA32 floating-point performance is currently abysmal. This is an area of current work; we hope to release a version with more reasonable performance in the near future.

Some classes of codes will not run fast on RVM. Known issues include:

- RVM start-up is slow compared to the IBM product JVM.
- Remember that the non-adaptive configurations (eg. Fast) opt-compile *every* method the first time it executes. With aggressive optimization levels, opt-compiling will severely slow down the first execution of each method. For many benchmarks, it is possible to test the quality of generated code by either running for several iterations and ignoring the first, or by building a warm-up period into the code. The SPEC benchmarks already use these strategies. The adaptive configuration does not have this problem; however, we cannot stipulate that the adaptive system will compete with the product on short-running codes of a few seconds.
- We expect RVM to perform well on codes with many threads, such as VolanoMark. However, if you have a code with many threads, each using JNI, RVM performance will suffer due to factors in the design of the current thread system.
- RVM does *not* yet support on-stack replacement for optimizing methods. The adaptive system will not optimize a single invocation of a long-running method.
- Performance on tight loops may suffer. The RVM thread system relies on quasi-preemption; the optimizing compiler inserts a thread-switch test on every back edge. This will hurt tight loops, including many simple microbenchmarks. We should someday alleviate this problem by strip-mining and hoisting the yield point out of hot loops.
- The thread system currently uses a spinning idle thread. If a RVM virtual processor (ie., pthread) has no work to do, it spins chewing up cpu cycles. Thus, RVM will only perform well if there is no other activity on the machine.
- The load balancing in the system is naive and unfair. This can hurt some styles of codes, including bulk-synchronous parallel programs.
- The default hash codes assigned by the system are currently only 8 bits. If you observed this as a problem, let us know in order to motivate us to fix this.
- The adaptive system may not perform well on SMPs; this may be due to bad interaction with the thread load balancer.

The RVM developers wish to ensure that RVM delivers competitive performance. If you can isolate reproducible performance problems, please let us know.

10 Coding Style Guidelines

This section describes a set of coding style guidelines that we recommend for all code added to the RVM system.

Regrettably, much code in the current system does not follow any consistent coding style. This unfortunate residue of the system's evolution make editing sometimes unpleasant, and prevent javadoc from formatting comments in many files. To alleviate this problem, we present this style guide (which consists of a small tweak of the style guide advanced by Sun) for new code.

Most code in the optimizing compiler has been formatted to at least obey the indentation rule, 80 columns, and javadoc comments. Most other RVM code has not been so formatted. We ask that all new code introduced into the system, at least for the optimizing compiler, follow the guidelines unless compelling factors dictate otherwise.

10.1 Coding style description

The RVM coding style guidelines are defined as a reference to an Sun Microsystems "Code Conventions for the Java Programming Language", with a few exceptions listed below. The Sun coding conventions can be found at `http://java.sun.com/docs/codeconv` in HTML, postscript, and PDF. Most of the style guide is intuitive; however, please read through the document or look at its sample code.

We have adopted one revision to the Sun code conventions:

1. **Two space indenting** The Sun coding convention suggests 4 space indenting, however with 80 column lines and 4 space indenting, there is very little room left for code. Thus, we recommend using 2 spaces indenting.

10.2 Javadoc requirements

All files should contain descriptive comments in Javadoc form (`http://java.sun.com/j2se/javadoc/index.html`) so that documentation can be generated automatically. Of course, additional non-javadoc source code comments should appear as appropriate. For javadoc, at a minimum,

1. All classes and methods should have a block comment describing them

2. All methods contain a short description of their arguments (using `@param`), the return value (using `@return`) and the exceptions they may throw (using `@throws`).

3. Each class should include `@see` and `@link` references as appropriate.

10.3 Useful tools/hints

This section describes helpful hints for conforming with the style guide. Below are suggestions on how to setup the two most common editors, emacs and vi.

10.3.1 emacs

The following tells `emacs` to indent 2 spaces:

```
;; You have to do it in this complicated way because of the
;; strange way the cc-mode initializes the value of ‘c-basic-offset’.
(add-hook 'c-mode-hook (lambda () (setq c-basic-offset 2)))
(add-hook 'java-mode-hook (lambda () (setq c-basic-offset 2)))
```

If you want emacs to truncate long lines instead of wrapping them, add the following to your c/java mode hook:

```
(setq truncate-lines 't)
```

10.3.2 vi

If you are more comfortable with vi, it is recommended that you use a vi clone called vim (http://www.vim.org). It contains all of vi's commands and is fully backward compatible, but is much more configurable than vi. Hints for vi diehards who absolutely refuse to use vim are provided at the end of this subsection (10.3.2).

vim Add the following to your .vimrc for formatting:

```
set shiftwidth=2           " for indenting and shifting
set expandtab              " to replace tab characters by spaces
set smarttab               " to allow the use of <Tab> for indenting
set formatoptions-=t2croq  " reset formatting
set formatoptions+=croq    " format comments
set textwidth=0            " don't wrap text
set wrapmargin=0           " ditto
" Java mode setup
augroup java
   autocmd!
   autocmd BufEnter *.java set cindent
   autocmd BufEnter *.java set cinoptions=>s,e0,n0,f0,{0,}0,^0,:s,=s,ps,ts,c3,+s,(0,u0,)20,*30,gs,hs
   autocmd BufEnter *.java set cinwords=if,else,while,do,for,switch,static,new
   autocmd BufLeave *.java set nocindent
augroup END
```

If you want vim to truncate long lines instead of wrapping them, add the following to your .vimrc:

```
set formatoption+=t " to allow autowrap text
set textwidth=74    " to allow autowrap text at 74th column
```

vi Standard vi options that would approximate Java formatting are:

```
set shiftwidth=2  " for indenting and shifting
set autoindent    " automatically indent new lines to the start of previous
```

and the approximation for wrapping long lines is

```
set wrapmargin=6  " to allow autowrap text at 74th column
```

11 FAQ

RVM Frequently Asked Questions

11.1 General

For most general Jikes RVM questions and answers, see `http://www.ibm.com/developerworks/oss/jikesrvm/info/overview.shtml`.

11.1.1 What is RVM?

The short answer: The Jikes Research Virtual Machine (RVM) is a software project designed to provide the academic and research communities with a flexible testbed that makes it possible to quickly prototype new virtual machine technologies and experiment with different design choices. It executes Java™[20] programs useful for research on fundamental virtual machine design issues. It runs on the AIX™[21]/PowerPC™[22], Linux©/PowerPC and Linux/IA-32, and exhibits industry-strength performance for many benchmark programs on the first two of these platforms. The RVM includes the latest VM technologies for dynamic compilation, adaptive optimization, garbage collection, thread scheduling, and synchronization.

11.1.2 Who is using RVM?

A list of current RVM researchers is available at `http://www.ibm.com/developerworks/oss/jikesrvm/info/users.shtml`. If you would like to be added to the web page, let us know.

11.1.3 Can I use RVM when teaching a class?

Yes, this is fine under the Common Public License

11.1.4 Who can I contact with questions?

Use the mailing lists available at `http://www-124.ibm.com/developerworks/projects/jikesrvm`

11.1.5 Which mailing list(s) should I subscribe to?

We currently have the following four mailing lists:

jikesrvm-researchers General discussion of Jikes RVM design, implementation, issues, and plans.

jikesrvm-regression Automatic mail messages and subsequent discussion regarding nightly regression runs.

jikesrvm-announce Infrequent announcements and news items.

jikesrvm-core Discussion of day-to-day development and design among JikesRVM core team members.

---

[20] **Java** and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

[21] **AIX** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

[22] **PowerPC** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

11.1.6 How can I contribute to RVM?

Bug reports or feature requests can be submitted directly at `http://www-124.ibm.com/developerworks/projects/jikesrvm`. We have not worked out a formal process for code contributions just yet, but hope to have it in place soon. We expect to follow the model of other successful open-source projects. For now, if you'd like to contribute code, let us know.

Word to the wise potential contributor: we cannot accept any contributions that infringe on anyone's intellectual property rights or copyrights.

11.2 Getting RVM and Documentation

11.2.1 How do I get RVM?

You need to download two bundles: the RVM source, and the RVM standard library jar file. Each of these is available for download from DeveloperWorks at `http://www-124.ibm.com/developerworks/projects/jikesrvm`. The RVM source is also available through a public CVS server.

You can also download the source to the libraries under a separate license.

11.2.2 Is there a list of known bugs?

See the bug tracking system available on DeveloperWorks at `http://www-124.ibm.com/developerworks/projects/jikesrvm`.

The bug tracking lists *defects*, representing bugs in the system, and *features*, which are TODO items to improve the system.

11.2.3 Is there documentation on-line?

Yes. See the RVM Home page at `http://www.ibm.com/developerworks/oss/jikesrvm`.

11.2.4 Can I get the Quicksilver Quasi-Static System?

No. This project is no longer active or supported.

11.2.5 Can I get DejaVu?

Not yet. The DejaVu team hopes to release some code shortly.

11.3 Building RVM

11.3.1 Which jikes should I use?

At Watson, we're currently using `jikes` v1.13 to compile the RVM source on both Linux and AIX™[23]. We've had reports from users that v1.14 has problems on Linux. In order to build the `rvmrt.jar` library, we applied patch 62 to the jikes build to fix a jikes scoping problem.

[23] **AIX** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

11.3.2 Has anybody thought about incremental boot image writing?

Incremental boot image building is not a trivial problem. One big issue is: if we change the implementation of one class in the boot image, what other parts of the VM image must be invalidated? One example: which methods must be recompiled to reflect the new implementation? We have no mechanism in place to trace these kinds of dependencies. There are other examples, too. In summary: incremental boot image writing would be nice, but it's not easy to support, and it hasn't been at the top of our priorities.

11.3.3 Can I recompile the RVM classes without rebuilding the boot image?

Yes. Use `jbuild -nolink`. This will copy, preprocess, and compile the RVM source files, but will not re-link the boot image. This is extremely useful for development, when the modified classes do not go in the boot image.

11.3.4 Can I monitor progress during jbuild?

Yes. Use `jbuild -trace` to see detailed progress.

11.3.5 How can I include my own classes in the boot image?

The `jconfigure` script defines which classes go in the boot image, by spitting out the file $RVM_BUILD/RVM.primordials. By default, any class with the `VM_` prefix in a defined directory set, goes in the boot image.

You may choose to add more classes to the primordial list. One way to do this is to edit `jconfigure`; look at the function `emitImageLinker`. You will see that the script already puts certain other non-VM classes in the primordial list (eg. `java.lang.Object`).

11.4 Runtime implementation

11.4.1 Does RVM have an interpreter?

No. RVM relies on two compilers, and compiles all methods to native code.

11.4.2 Does RVM support JNI?

Most JNI functionality is supported. A few functions are not. Some functions are supported on AIX™[24], but not Linux.

11.4.3 Does RVM support user-defined class loaders?

No. Some class loader functionality is hacked around, delegating all function to the system class loader.

11.4.4 Does RVM support the Java security model?

No.

11.4.5 Does RVM support serialization?

Sort of. RVM provides enough serialization support to run some codes. However, serialization of classes in our class library may not match serialization as implemented in the Sun class library.

[24] **AIX** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

11.4.6 Does RVM enforce the Java Memory Model?

No. Depending on the architecture, various features of the memory model are not implemented according to the current spec, to the best of our understanding.

Known issues include:

- on PowerPC™[25], the system does not enforce sequential consistency for volatile variables
- the system does not enforce atomicity of memory accesses for doubleword values
- by default, the optimizing compiler does not respect the "reads kill" property. However, there is a command-line option to enforce the property, which constrains the optimizations.

11.4.7 Does RVM support zip files?

Sort of. RVM includes enough zip support to read compressed jar files. For complete java.util.zip functionality, you are recommended to find a pure Java implementation of java.util.zip. We have successfully used JazzLib.

11.4.8 Why doesn't the RVM source use packages?

This is a historical artifact. In the early days of the project, we did not want to be constrained by a package structure to a particular directory structure. With the current build process, this is not an issue, and we may incrementally add package structure to the implementation over time.

11.4.9 How do RVM's threads, Posix threads, and kernel threads relate to each other?

RVM implements an m-to-n threading model, where m is the number of Java threads and n is the number of Posix threads (ie., pthreads). RVM does not know or care whether the Posix threads are implemented as kernel threads or user-level level threads. You can specify n, the number of Posix threads to use, on the command line with `-X:processors=n`. You should normally set n to be the number of physical processors on your machine.

In the source code, a `VM_Thread` is the base class for each Java thread, and a `VM_Processor` is the base class representing each Posix thread.

11.4.10 What are the semantics of VM_Uninterruptible?

The actual semantics of uninterruptibility are: if a class extends `VM_Uninterruptible` , then the compiler will not generate yield points in methods of the class. So, there will be no timer-driven thread switches caused by SIGALARMs in these methods. Additionally, the compiler will *not* check for stack overflow in the method prologue. So, an uninterruptible method will never cause a stack overflow trap.

You should exercise extreme caution when modifying uninterruptible code. It is generally not legal (although not enforced) to throw an exception or cause GC from an uninterruptible method. Uninterruptible code should not call interruptible code. Furthermore, any uninterruptible method should not need more stack space than specified in VM_StackFrameLayoutConstants for the stack "guard" area.

It is perhaps unfortunate that the uninterruptible attribute is currently specified on a class level. In some cases, we have declared an entire class uninterruptible, even though only a few methods of the class are really not safe to interrupt. We would like to someday use a finer-grain mechanism by which to annotate individual methods as uninterruptible.

[25] **PowerPC** is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

11.4.11 What is the list of operations that may cause a GC?

Any operation that allocates memory or causes memory to be allocated may force a GC. Some cases to look out for include:

- any instruction that throws an exception,
- any call that may cause a stack overflow,
- any monitorenter on a contended lock,
- string concatenation, and
- any thread-switch point may allow another thread to force GC.

11.4.12 How can I implement a new GC algorithm?

See the memory management section of the userguide.

11.4.13 How does RVM enter native code?

There are two mechanisms whereby RVM may transition from Java to native code.

The first mechanism is when RVM calls a method through a VM `syscall` method. These native methods are non-blocking system calls or C library services. To implement a syscall, the RVM compilers generate a call sequence consistent with the platform's underlying calling convention. A `syscall` is not a GC-safe point, so `syscalls` may modify the Java heap (eg. memcpy).

The second mechanism is JNI. Naturally, the user writes JNI code using the JNI interface. RVM implements a call to JNI with a special JNI compiler that generates a stub routine stack frame that manages the transition between Java and native code. The thread system implements recovery mechanisms to deal with JNI methods that block or otherwise fail to return to Java promptly. A JNI call is a GC-safe point, since JNI code cannot freely modify the Java heap.

11.4.14 What happens to thread switching while a thread is executing native code?

There are two ways to execute native code: `syscalls` and JNI. A Java thread that calls native code by either mechanism will never be preempted by RVM. As far as RVM is concerned, a Java thread that enters native code 'owns' the underlying `VM_Processor` (pthread) until it returns to Java. Of course the OS may preempt the underlying pthread; this falls beyond RVM's control.

Some activities (eg. GC) require all threads currently running Java to halt. So what happens when one Java thread forces a GC while another Java thread is executing native code?

If the native code is a `syscall`, then the VM stalls until the native code returns. Thus, all `syscalls` should be non-blocking operations that return fairly soon. Note that a `syscall` is *not* a GC-safe point.

If the native code is JNI (outside RVM control), then the thread system will wait for a while, eventually declare the underlying pthread "out to lunch", and continue execution with the remaining pthreads. Note that JNI code is a GC safe point; non-malicious correct native code cannot perturb the Java heap without notifying the RVM through a JNI method invocation. Hopefully the userguide will soon describe this in more detail.

See section 4.6 for more details on the thread system.

11.4.15 How do the various locking and synchronization mechanisms relate to each other?

There are at least six ways to enforce mutual exclusion in the RVM runtime. For normal library code and most VM code, monitorenter and monitorexit should suffice. The lower-level primitives provide building blocks for implementing monitorenter and exit. Some VM systems, such as thread scheduling and GC, resort to lower-level primitives for situations where normal Java object locking is inconvenient or illegal.

VM_Magic.prepare and VM_Magic.attempt The RVM compiler translates these VM_Magic calls into low-level hardware-supported atomic sequences. These low-level primitives are the building blocks for all other mutual exclusion mechanisms.

The prepare call fetches the contents of a memory location and begins a conditional critical section. The attempt call ends the conditional critical section, and returns true if and only there were no intervening writes to the guarded memory location.

On PowerPC, the compilers implement prepare and attempt using the lwarx and stwcx instructions. On IA32, the compilers rely on CMPXCHG with the LOCK prefix.

VM_Synchronization This class implements some useful common low-level synchronization sequences, such as fetch-and-add and test-and-set. The VM_Synchronization primitives, in turn, are implemented using VM_Magic.prepare and attempt.

VM_ProcessorLock This lock is used to enforce mutual exclusion between `VM_Processors` (pthreads.) It provides a non-blocking attempt to require the lock (`tryLock()`) as well as a blocking spin-lock (`lock()`).

VM_Lock This class provides the normal synchronization operations on Java objects between Java threads. The implementation is a variant of Thin Locks.

monitorenter and monitorexit Synchronized statements in Java source code are compiled to monitorenter and exit in the Java bytecode. The RVM compilers implement these bytecodes by inserting calls to `VM_Lock` routines; the optimizing compiler inlines the common cases.

VM_GCLocks This class simply encapsulates a number of VM_Synchronization locks used for various purposes by the RVM GC system.

11.4.16 What causes VM_JNIEnvironment.setNames() to run?

This function, like many others in the RVM, is called during the RVM boot sequence.

11.4.17 Does RVM conform to Sun's JDK Host Porting Interface?

No. There is nothing in RVM that remotely resembles HPI.

11.5 Libraries

11.5.1 Does RVM run awt?

Not currently. We've tried this recently; RVM does not currently provide all the JNI support needed for awt. We have not looked into how difficult it would be to provide the missing JNI support.

11.5.2 Can I run some standard library on RVM that is not included in rvmrt.jar?

You can try. Set your classpath to pick up the library you desire.

11.6 Optimizing Compiler

11.6.1 What is a PEI?

PEI is our acronym for potentially excepting instruction. This applies to any instruction in the IR that may throw an exception.

11.6.2 What is AOS?

AOS stands for adaptive optimization system.

11.6.3 Is there a difference between a GC safe point and a thread switch point?

Yes. Every thread switch point is a GC safe point, but every GC safe point need not be a thread switch point.

A thread switch point is an instruction where the RVM thread scheduler may intervene and cause a different Java™[26] thread (VM_Thread) to run on the current pthread (VM_Processor), even if no exception is thrown. Thread switch points include yield points inserted in prologues, epilogues, and back edges, monitorenter and exits.

A GC safe point is any instruction where the compiler must generate a GC map, including every thread switch point. In particular, every PEI is a GC point.

11.6.4 How do I find the def of a register in SSA form?

Use `OPT_Register.getFirstDef()`.

If this returns `null`, then either a) the register is dead and it's definition has been eliminated, or b) the def-use chains are not up-to-date.

The def-use chains are not normally kept up-to-date incrementally. To recompute the def-use chains, call `OPT_DefUse.computeDU(ir)`. Most optimization passes over SSA form call this method at the beginning of the compiler phase.

11.6.5 What is Heap Array SSA form?

See the SAS 2000 paper, as well as comments in `OPT_SSA.java`.

11.6.6 Is ABCD included?

The open-source distribution includes a derivative of the prototype ABCD implementation used for the PLDI 2000 paper. However, ABCD is *not* enabled by default. The current implementation is incorrect, as it checks only upper bounds and not lower bounds. There is a command-line option to turn on the current implementation.

11.6.7 Is escape analysis included?

The interprocedural flow-sensitive escape analysis in the OOSPLA 99 paper is not currently included.

However, the distribution includes a less-powerful flow-insensitive escape analysis. See `OPT_SimpleEscape.java`.

11.6.8 How do I insert my new compiler pass in the optimizing compiler driver?

See section 5.2 of the userguide, which describes how to add phases to class `OPT_OptimizationPlanner`.

[26] **Java** and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

11.6.9 What if I want my pass to do inter-procedural analysis?

The normal RVM does not have a convenient entrypoint for IPA. As Java is a dynamic language, the RVM continually compiles classes as they are loaded. Each method is compiled individually.

You can use the `OptTestHarness` driver to define a set of classes. This driver program loads a set of classes or methods defined on the command line. You can then add an entrypoint in `OptTestHarness.java` that calls your IPA after loading all the relevant classes.

If you come up with a general mechanism for this, please consider contributing it back.

11.6.10 What is the OptTestHarness?

The OptTestHarness (see section 5.8) is a driver program to run the optimizing compiler even on a BaseBase boot image. This driver is useful for optimizing compiler development, since you can use the driver to selectively compile individual methods with certain options.

Acknowledgements

Thanks to the following folks, who contributed text and/or bug fixes to this document.

- Bowen Alpern
- Matthew Arnold
- Dick Attanasio
- David Bacon
- Steve Blackburn
- Maria Butrico
- Anthony Cocchi
- John Davis
- Julian Dolby
- Stephen Fink
- David Grove
- Mike Hind
- Matthias Hauswirth
- Wilson Hsieh
- Mark Mergen
- Igor Pechtanski
- Barbara Ryder
- Stephen Smith
- Peter Sweeney
- Martin Trapp

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2000.

[3] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

12 Appendix A: RVM command-line directives

Currently, the RVM has two types of configurations: the adaptive configurations contain the adaptive optimization system (AOS), and the non-adaptive configurations that do not. This section describes the nonstandard RVM command-line directives that provide the mechanism to specify adaptive optimization system and optimizing compiler options. Section 12.1 describes how command-line directives are specified in a non-adaptive configuration, and Section 12.2 describes how command-line directives are specified in a adaptive configuration.

12.1 Command-Line Directives in Non-adaptive Configurations

This section describes how nonstandard RVM command line options are specified for a non-adaptive configuration of RVM. In a non-adaptive configuration, the command line options modifies the behavior of the initial runtime compile when this compiler is the optimizing compiler.

To pass an option to the optimizing compiler, the option is prefixed with `-X:irc:`. For example, to perform global array bounds check elimination on demand when a method is initially compiled with the optimizing compiler, use the `-X:irc:global_bounds=true` directive.

Please note that a particular non-adaptive configuration (such as OptBasecopyingGC) may not use the optimizing compiler as the initial runtime compiler, but rather use the baseline compiler. Therefore, the validity of the prefix to access the optimizing compiler as the initial runtime compiler depends on how the configuration is built.

All of the above nonstandard VM directives must occur before the application class name and application's command-line options.

For the following discussion, we assume that the appropriate prefix has been prepended to the option and only discuss the option.

To see descriptions of command-line arguments to the optimizing system, use the **`help`** option; that is, `-X:irc:help`. As of this writing, this command produces the following output:

-X:irc[:help] Print brief description of opt compiler's command-line arguments
-X:irc:printOptions Print the current values of the active opt compiler options
-X:irc:O0 Select optimization level 0, minimal optimizations
-X:irc:O1 Select optimization level 1, modest optimizations
-X:irc:O2 Select optimization level 2
-X:irc:O3 Select optimization level 3

Boolean Options (-X:irc:<option>=true or -X:irc:<option>=false)

option	OptLevel	Description
local_cse	1	Perform local common subexpression elimination
local_constant_prop	1	Perform local constant propagation
local_copy_prop	1	Perform local copy propagation
local_sr	1	Perform local scalar replacement
local_check	1	Perform local elimination of redundant null checks and array bounds checks
global_bounds		Perform global Array Bound Check elimination on Demand
simple	1	Perform suite of simple flow-insensitive optimizations
monitor_removal	1	Try to remove unnecessary monitor operations
invokee_thread_local		Compile the method assuming the invokee is thread-local
simple_escape_ipa		Eagerly compute method summaries for simple escape analysis
field_analysis	0	Eagerly compute method summaries for flow-insensitive field analysis
scalar_replace_aggregates	1	Perform scalar replacement of aggregates
reorder_code	1	Reorder basic blocks for improved locality and branch prediction
inline_new	1	Inline allocation of scalars and arrays
inline	1	Inline statically resolvable calls
eager_inline	1	Speculatively inline non-final virtual calls
eager_inline_interface	1	Speculatively inline non-final interface calls
preex_inline	1	Pre-existence based inlining
ssa	2	Should SSA form be constructed on the HIR?
pruned_ssa		Use liveness information to reduce insert phi nodes?
semi_pruned_ssa		Use local approximation of liveness information to reduce insert phi nodes?
load_elimination	2	Should we perform redundant load elimination during SSA pass?
coalesce_after_ssa	2	Should we coalesce move instructions after leaving SSA?
expression_folding	2	Should we try to fold expressions with constants in SSA form?
store_elimination		Should we perform dead store elimination during SSA pass?
value_prop		Perform value propagation
gcp	2	Perform global code placement
gcse	2	Perform global code placement
verbose_gcp		Perform noisy global code placement
value_prop_verbose		Perform noisy value propagation
unwhile	2	Turn whiles into untils
handler_liveness	2	Store liveness for handlers to improve dependence graph at PEIs
schedule_prepass		Perform prepass instruction scheduling
errors_fatal		Should an unexpected failure during compilation be treated as fatal?
no_checkcast		Should all checkcast operations be (unsafely) eliminated?
no_checkstore		Should all checkstore operations be (unsafely) eliminated?
no_bounds_check		Should all bounds check operations be (unsafely) eliminated?
no_null_check		Should all null check operations be (unsafely) eliminated?
no_synchro		Should all synchronization operations be (unsafely) eliminated?
no_threads		Should all yield points be (unsafely) eliminated?
no_cache_flush		Should cache flush instructions (PowerPC SYNC/ISYNC) be omitted? NOTE: Cannot be correctly changed via the command line!
detect_uniprocessor		Does the system automatically detect when there is one PHYSICAL CPU, and optimize accordingly? NOTE: Cannot be correctly changed via the command line!
reads_kill		Should we constrain optimizations by enforcing reads-kill?

monitor_nop	Should we treat all monitorenter/monitorexit bytecodes as nops?
static_stats	Should we dump out compile-time statistics for basic blocks?
loop_depth_edge_counts	Use loop depth to approximate edge counts
fixed_jtoc	Exploit knowledge that the value of JTOC is a constant? (IA32 only)
phases	Print short message for each compilation phase
verbose	Print method name at start of compilation
print_inline_report	Print detailed report of compile-time inlining decisions
dom	Print dominators
pdom	Print post-dominators
print_ssa	Print SSA form
print_dg_burs	Print dependence graph before burs
print_dg_sched_pre	Print dependence graph before prepass scheduling
print_dg_sched_post	Print dependence graph before postpass scheduling
pcoal	Print coalescing output
print_detailed_inline_report	Print report regarding inlining decisions made with Gnosys turned on
high	Print IR after initial generation
final_hir	Print IR just before conversion to LIR
low	Print IR after conversion to LIR
final_lir	Print IR just before conversion to MIR
mir	Print IR after conversion to MIR
final_mir	Print IR just before conversion to machine code
mc	Print final machine code
cfg	Print control flow graph too when IR is printed
print_schedule_pre	Print IR after prepass scheduling
print_schedule_post	Print IR after postpass scheduling
regalloc	Print IR before and after register allocation
print_calling_conventions	Print IR after expanding calling conventions
vcg_dg_burs	Dump dependence graph before BURS in vcg form
vcg_dg_sched_pre	Dump dependence graph before prepass scheduling in vcg form
vcg_dg_sched_post	Dump dependence graph before postpass scheduling in vcg form
cgd	Enable debugging support for final assembly
debug_abcd	Enable debugging support for ABCD (Array Bound Check elimination on Demand)

Value Options (-X:irc<option>=<value>)

option	Type	Description
ic_max_target_size	int	Static inlining heuristic: Upper bound on callee size
ic_max_inline_depth	int	Static inlining heuristic: Upper bound on depth of inlining
ic_max_always_inline_target_si	int	Static inlining heuristic: Always inline callees of this size or smaller
ic_max_inline_expansion_factor	int	Static inlining heuristic: Upper bound on relative expansion of caller due to inlining
ic_max_method_size	int	Static inlining heuristic: Upper bound on absolute size of caller due to inlining
ai_max_target_size	int	Adaptive inlining heuristic: Upper bound on callee size
ai_max_inline_expansion_factor	int	Adaptive inlining heuristic: Upper bound on relative expansion of caller due to inlining
ai_max_method_size	int	Adaptive inlining heuristic: Upper bound on absolute size of caller due to inlining
max_deadstore_system_size	int	Maximum number of dataflow equations for dead store elimination
unroll_log	int	Unroll loops. Duplicates the loop body n times.
cond_move_cutoff	int	How many extra instructions will we insert in order to remove a conditional branch?
load_elimination_rounds	int	How many rounds of redundant load elimination will we attempt?
alloc_advice_sites	String	Read allocation advice attributes for all classes from this file

Selection Options (set option to one of an enumeration of possible values)

Selection of register allocation algorithm

regalloc_algorithm	ls

```
Selection of guard mechanism for inlined virtual calls that cannot be statically bound
inlining_guard               ig_method_test ig_class_test ig_trap
Selection of strictness level for floating point computations
fp_mode                      strict allow_fma allow_assoc
Selection of spilling heuristic
spill_cost_estimate          simple brainDead loopDepth
Selection of mechanism to clean IA32 FP stack at procedure boundaries
fclear                       fninit fstp ffree
Selection of mechanism to convert double to int on IA32
f2int                        VM_Math fist

Set Options (option is a set of values)
exclude                       Exclude methods from being opt compiled
method_to_print               Only print IR from methods whose name contains this string
```

The `printOptions` command-line optimizing compiler option will print the current setting of optimizing compiler's options. Please note that the order of the `printOptions` command-line directive with respect to other optimizing compiler command-line directives is important. When a `printOptions` directive is found, the setting of the optimizing compiler options will reflect only those optimizing compiler directives that have preceded the `printOptions` directive.

12.2 Command-Line Directives in Adaptive Configurations

This section describes how nonstandard RVM command line options are specified for an adaptive configuration. In an adaptive configuration, the command line options modifies the behavior of either the adaptive optimization system or the optimizing compiler. A command line directive is constructed by concatenating an option with a prefix which identifies the desired destiny for that option.

All options in an adaptive configuration are prefixed with `-X:aos`. To pass an option to the adaptive optimization system, use the `-X:aos:` prefix. For example, to set the logging level of AOS to one, use the directive `-X:aos:logging_level=1`. Unlike a nonadaptive configuration, an adaptive configuration may conceptually have many optimizing compilers that are available at runtime, each with its own set of option values. We present a mechanism to address each conceptual optimizing compiler. To pass options to the opt compiler that recompiles a methods use the `-X:aos:opt[?]` prefix where the `?` is optional and if specified is an integer that identifies the optimization level. For example, `-X:aos:opt2:global_bounds=true` performs global array bounds check elimination on demand when a method is optimized at optimization level 2. If no optimization level is specified, the option applies to all optimization levels of the optimizing compiler that recompiles methods. For example, `-X:aos:opt:global_bounds=true` performs global array bounds check elimination on demand whenever a method is recompiled with optimization. Like a nonadaptive configuration, the optimizing compiler may be used as the initial runtime compiler in an adaptive configuration. If so, options are passed to the initial runtime compile by prefixing each option with `-X:aos:irc:`. For example, to perform global array bounds check elimination on demand when a method is initially compiled with the optimizing compiler, use the `-X:aos:irc:global_bounds=true` directive. See Section 12.1 for a discussion of the optimizing compiler command-line options that are available.

Please note that a particular adaptive configuration may not use the optimizing compiler as the initial runtime compiler, but rather use the baseline compiler. Therefore, the validity of the prefix to access the optimizing compiler as the initial runtime compiler depends on the configuration.

Finally, the prefix `-X:aos:share[?]:o=v` is a short hand for passing a option value pair, `o=v`, to both the AOS and to the different "conceptual" optimizing compilers. If the optimization level is specified, then only that optimization level for recompilation is affected. Otherwise, the option is set for the initial runtime optimizing compiler and all optimization levels for method recompilation.

All of the above nonstandard VM directives must occur before the application class name and application's command-line options.

For the following discussion, we assume that the appropriate prefix has been prepended to the option and only discuss the option.

12.2.1 Adaptive Optimization System (AOS) Command-Line Directives

To see a description of the command-line directives to the AOS, use `-X:aos:help`. As of this writing, this command produces the following output:

```
-X:aos[:help]       Print a brief description of AOS command-line options
-X:aos:printOptions Print the current option values of AOS
-X:aos:o=v          Pass the option-value pair, o=v, to AOS
-X:aos:opt[:help]   Print a brief description of optimizing compiler's command-line options that recompiles a method
-X:aos:irc[:help]   Print a brief description of optimizing compiler's command-line options that initially compiles a method
-X:aos:share[?]:o=v Pass the option-value pair, o=v, to AOS and to the optimizing compiler.  If the optimization level is specified, then only that optimization level is affected when recompiling a method

Boolean Options (-X:aos:<option>=true or -X:aos:<option>=false)
option                               Description
gather_profile_data                    Should profile data be gathered and reported at the end of the run?
adaptive_inlining                      Should we use adaptive feedback-directed inlining?
adaptive_recompilation                 Should we make recompilation decisions adaptively?
dump_ai_decisions                      After completion, should we dump inlining decisions?
use_offline_inline_plan                Should we read a inlining plan from disk and use it to perform inlining?
hardware_performance_monitors          Should we use the hardware performance monitors?
use_compiler_dna_file                  Should we read the compiler DNA (relative cost/benefits) from a file?
report_static_program_stats            On exit, print some static summary information of the program?
insert_yieldpoint_counters             Should we insert instrumentation to count yieldpoints executed?
insert_method_counters_opt             Should Intrusive method Counters be enabled for opt-compiled code?
insert_instruction_counters            Insert counters on all instructions and report counts at end.
insert_debugging_counters              Allows easy insertion of counters to help debug OPT compiled code.

Value Options (-X:aos<option>=<value>)
option                         Type    Description
initial_sample_size            int     Initial value of how many samples to take before reporting method hotness to controller
max_opt_level                  int     What is the highest optimization level to use when adaptive recompilation is employed?
default_opt_level              int     The optimization level to use for recompilation (with simple strategy)
filter_opt_level               int     At what optimization level should we begin to NOT return to methods to the controller
mso_adjust_bounds              double  Samples adjusted by window-history MSO by factors of 1.0 +/- MSO_ADJUST_BOUNDS
mso_num_epochs                 int     How many epochs to maintain for the window-history based MSO
lf                             String  Name of log file
ilf                            String  Name of inline decision log file
dna                            String  Name of compiler DNA file
logging_level                  int     Control amount of event logging (larger ==> more)
final_report_level             int     Control amount of info reported on exit (larger ==> more)
sample_freq_millis             double  Sample frequency, in milliseconds
dark_matter                    double  What percentage of execution time, for one reason or another, does not appear in samples?
decay_frequency                int     After how many clock ticks should we decay
decay_rate                     double  What factor should we decay by
ai_decay_rate                  double  What factor should we decay by AI hotness by
fixed_recompilation_overhead   double  Fixed cost of recompiling and installing a method
ai_sample_size                 int     After how many samples do we update the weights in the dynamic call graph?
initial_ai_threshold           double  What fraction (0.0 to 1.0) of total dynamic calls makes an edge hot, at program startup?
final_ai_threshold             double  What fraction (0.0 to 1.0) of total dynamic calls makes an edge hot, in steady state?
ai_method_hotness_threshold    double  What fraction (0.0 to 1.0) of total method samples makes a callee hot enough to consider for recompilation to inline a hot non-inlined call edge?
max_expected_ai_boost          double  How much more effective do we expected an optimization level to be if boosted by adaptive inlining?
offlinePlan                    String  Name of offline inline plan to be read and used for inlining
controller_input_queue_size    int     The size of the controllerInputQueue
compilation_queue_size         int     The size of the compilation queue

Selection Options (set option to one of an enumeration of possible values)
Selection of primary controller strategy
```

```
primary_strategy              baseonly quickonly optonly adaptive
Selection of method sample (recompilation) organizer
method_sample_organizer       window windowHist
```

Set Options (option is a set of values)

The `primary_strategy` option determines what strategy is used to compile methods. The default strategy is `adaptive` which allows a method to be recompiled multiple times at different optimization levels. The other strategies allow an adaptive configuration to behave as a just-in-time compiler (JIT) by determining what compiler will compile a method. For example, the `optonly` strategy compiles a method once with the optimizing compiler. To optimize compile a method at optimization level 1, use the `-X:aos:irc:O1` option. (Note that to obtain the functionality of a JIT at optimization level 1, `-X:primary_strategy=optonly -X:aos:irc:O1`, in a nonadaptive configuration would be achieved with the command-line option `-X:irc:O1`.) Currently, the `quickonly` primary strategy is not supported.

Another interesting combination of command-line option that allow single level adaptive recompilation is `-X:aos:adaptive_recompilation=false -X:aos:default_opt_level=1` which will recompile hot methods at optimization level 1 only.

12.2.2 Implementing AOS command-line options

This section provides some information on various implementation details for AOS command-line options.

The command-line options to AOS are stored as fields in an object of type `VM_AOSOptions`. The RVM build process generates the `VM_AOSOptions.java` file automatically from a template.

To add or modify the command-line options in `VM_AOSOptions.java`, you must modify either `BooleanOptions.dat`, `ValueOptions.dat`, `ShareBooleanOptions.dat`, or `ShareValueOptions.dat`. You should describe your desired command-line option in a format described below in Section 5. The options in the `ShareBooleanOptions.dat` and `ShareValueOptions.dat` files are defined as both AOS and optimizing compiler options.

12.3 Adding RVM Nonstandard Command-Line Options

This section states how nonstandard RVM command-line options can be added to RVM. Nonstandard RVM command-line options are those options that are specific to RVM. The format of a nonstandard RVM option is `-X:o=v` where `o` is the option and `v` is the value it is to be set to. Adherence to this format is important to keep command-line options processing from becoming unwieldy.

RVM command-line options are processed in two places: `RunBootImage.C` and `VM_CommandLineArgs.java`. `RunBootImage.C` is called first before the RVM boot image is loaded, and `VM_CommandLineArgs.java` is called after. In addition to processing any option which does not require RVM boot image to be loaded (such as `help` and `version`), `RunBootImage.C` processes any nonstandard option that impacts either heap size, message output, or where to find the boot image. To allow unrestricted RVM option orders and since command line processing stops at the first option that is not recognized as a RVM option, all other options must be recognized by `RunBootImage.C` and passed on.

Index

adaptive configurations, 9
AOS command-line options, 72
array access, 21

BooleanOptions.dat, 30, 72
boot image, 5, 8, 52
boot image compiler, 8
breakpoints, 43, 46
build directory, 5
BURS, 35

callbacks, 28
calling conventions, 24, 25
class initializer, 27
class loading, 26
command-line arguments, 40
command-line options, 12, 30
compilation, 32
concurrent garbage collection, 40
configuration names, 8
configurations, 5, 7, 43, 52
conflict resolution stub, 23
constant pool, 26
constants, 22

debugging, 42
deferred compilation, 23
dynamic type checking, 22

editing source code, 54
emacs, 54
environment variables, 6

field access, 21
finalizable objects, 40
finalizer method, 40

garbage collection, 39, 52

hash codes, 53
hashing, 21
HIR, 32

IMT, 23
indenting, 54
instruction selection, 35
InstructionFormatList.dat, 34
instructionFormats.java, 33
instructions, 33
interface methods, 23
interfaces, 22
IR, 32, 33

javadoc, 54
jbuild script, 8
jconfigure script, 7, 39, 43
jdp, 42
jikes, 12
JNI, 47
JTOC, 22, 43

lazy method compilation, 23
lazy method invocation stub, 23
LIR, 32
literals, 22
locking, 21, 27

magic methods, 37
methods, 22
MIR, 32

NullPointerException, 21

object header, 21, 40
on-stack replacement, 53
OperatorList.dat, 33
operators, 33
OPT_CompilationPlan class, 32
OPT_CompilerPhase class, 32
OPT_Operators class, 33
OPT_OptimizationPlanElement class, 32
OPT_OptimizationPlanner class, 32
OPT_Options class, 30
optimization plan, 32
optimizations, 32

PATH, 6

quasi-preemption, 53

register conventions, 24, 25
Remote Reflection, 42
runtime compiler, 8
rvm script, 12
RVM_BUILD, 6, 8
RVM_ROOT, 6

scheduling, 27
semantic inlining, 37
stack conventions, 24, 25

static methods, 23
stop-the-world garbage collection, 39, 40
superclass, 22
symbol map, 43

threads, 27
TIB, 22

ValueOptions.dat, 30, 72
vi, 54
virtual methods, 22
VM_Allocator class, 39, 40
VM_AOSOptions class, 72
VM_CollectorThread class, 40
VM_GCWorkQueue class, 40
VM_Handshake class, 40
VM_Magic, 37
VM_Method class, 32
VM_WriteBuffer class, 40

write barrier, 40
write buffer, 40