The Jikes^{TM*}Research Virtual Machine User's Guide v2.2.0

December 18, 2002

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

Contents

List of Figures

1 Introduction

This section provides an overview of the Jikes RVM as well as information for how best to use this document.

1.1 Welcome to the Jikes RVM

The Jikes$^{\mathrm{TM}}$[1] RVM is a Research Virtual Machine developed at the IBM T.J. Watson Research Center. Key features of the system include

- the entire virtual machine (VM) is implemented in the Java$^{\mathrm{TM}}$[2] 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 the Jikes RVM (formerly known as Jalapeño) appears in our published papers. For overviews of the system's structure, including the runtime system, optimizing compiler, and adaptive systems, see the published papers available from the Jikes RVM web page:

http://www.ibm.com/developerworks/oss/jikesrvm/info/pubs.shtml

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.

We have given several tutorials on Jikes RVM that you may find useful. The PACT'01 tutorial covers all of Jikes RVM. The PLDI'02 and OOPSLA'02 tutorials focus on the optimizing compiler. The tutorial slides are available at

http://www.ibm.com/developerworks/oss/jikesrvm/info/presentations.shtml

The Jikes RVM is a bleeding-edge research project. You will find that some of the code 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.

Many academic groups have adopted the Jikes RVM as their primary research infrastructure, resulting in publications in leading conferences, such as PLDI, POPL, OOPSLA, and SIGMetrics. In the first year of its open source release, the VM has been downloaded by over 1500 unique IP addresses, including over ninety universities around the world. A list of some of the users is available. Teaching resources using Jikes RVM are also available.

---

[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.

1.2 About this document

This document provides Jikes 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 Jikes RVM distribution, focusing on implementation details of how to build, run, and add functionality to the system.

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`. Each released Jikes RVM tarball holds a postscript version of this document as `userguide.ps` in the root tarball directory.

The RVM web page includes javadoc API documentation. This HTML should be the primary reference for individual classes. The level of detail provided in the javadoc is highly variable, but most Jikes RVM classes have at least a minimal description.

You may find sections of this user's guide missing, incomplete or otherwise confusing. 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 Jikes RVM researchers mailing list. Constructive criticism will be cheerfully accepted.

2 Installation Guide

So you've downloaded the Jikes RVM distribution. Now what? This section gives instructions on how to install and run the system.

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 either version 1.13 or version 1.18. We have seen problems with version 1.14 through 1.17. For releases of Jikes RVM before version 2.2.0, you must use jikes 1.13.
- 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.
- unzip. You can download an `unzip` implementation from `http://www.info-zip.org/pub/infozip/UnZip.html`.

2.1.1 AIX/PowerPC

In addition to the software mentioned above, to install, build, and run the Jikes 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`.
- a C/C++ compiler. We recommend `gcc v2.95.3`; it's been reported that other gcc versions on AIX don't work. It's also been reported that IBM's `xlC` works.

2.1.2 Linux/IA32

In addition to the software mentioned above, to install, build, and run the Jikes 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.
- a JDK. We currently recommend the Blackdown VM, v1.3.1. You can download this from `http://www.blackdown.org`. We have seen problems using the IBM 1.3.1 DK; avoid it until these problems can be resolved.
- `ksh`. This must be installed in `/bin/ksh`. If you don't already have a ksh, we recommend `pdksh`.

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

2.2 Installation Overview

To install and build the Jikes RVM, three items are required

- The Jikes RVM source distribution. This is available as a compressed tar file `jikesrvm-2.2.0.tar.gz`. You can also work with the contents of this repository with CVS from the public repository.
- The Jikes RVM standard libraries. This is a jar file `jlibraries-2.2.0.tar.gz`.
- The Classpath libraries.

Each item is distributed under a different license. The licenses for the first two items are provided in Appendix B. The GNU Classpath license is available at `http://www.gnu.org/software/classpath`.

The first and second items are available from the Jikes RVM download page. The third item is available at `http://www.gnu.org/software/classpath`. You do not always need to explicitly download the second and third items; the Jikes RVM build process will automatically check that these two items are present. If they are not present, and if the build process can confirm that you have viewed the appropriate license for JLIBRARIES, the build process will automatically download these items after asking for your confirmation.

Once these files have been downloaded, you will set up a working directory holding the Jikes RVM source files, standard library jar, and tools needed to build Jikes RVM.

Jikes RVM can be configured in various ways. Multiple versions of the system, corresponding to different configurations, can be generated from one working directory. See Section 2.4 for information about the various configurations. The Jikes 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.

To install Jikes RVM you must do the following:

1. Set up a working directory.
2. Set various environment variables.
3. Edit Jikes 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 Jikes 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 Jikes 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 Jikes RVM standard libraries. If you have previously downladed some version of the libraries, you may ignore this step and it will happen automatically for you when you try to build a Jikes RVM configuration. 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 Jikes 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

   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
   ```

   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 Jikes 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 Jikes RVM, they have probably already done this step for you. Consult your local Jikes 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.

[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.

The host and target configuration files have two sections. In the first section, you specify the operating system, architecture, and whether or not the platform will support SMP-builds of Jikes RVM. For operating system, define one of RVM_FOR_LINUX or RVM_FOR_AIX to be 1. For architecture define either RVM_FOR_IA32 or RVM_FOR_POWERPC to be 1. For SMP status, set RVM_FOR_SINGLE_VIRTUAL_PROCESSOR to 0 (SMP supported) or 1 (SMP not supported). The following are the typical settings for RVM_FOR_SINGLE_VIRTUAL_PROCESSOR

`AIX/PowerPC` 0

`Linux/PowerPC` 1

`Linux/IA32` 1 with a 2.2 kernel; 0 with a 2.4 kernel and glibc compiled to use the GS segment register to access pthread-specific state.

The second section in the configuration file is used to define how to find tools that Jikes RVM needs. You must set the following variables:

`HOST_JAVA_HOME` the base directory for JDK JVM. If you have a nonstandard JDK, you may have to define a number of variables that are normally defined in terms of `HOST_JAVA_HOME`.

`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 Jikes 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 Jikes RVM configuration.

   For quick turnaround time while modifying Jikes RVM, you will usually want to use the baseline compiler to build the Jikes RVM bootimage. A typical configuration that builds very quickly (but performs poorly) is `BaseBaseSemiSpace`: a non-adaptive system that uses the baseline compiler everywhere, with the semispace copying collector.

   To obtain reasonable performance from Jikes RVM, you will need to use the optimizing compiler to build the bootimage. This takes longer, but generates a Jikes RVM image with reasonable runtime performance. We recommend either the `FullAdaptiveSemiSpace` or `FastAdaptiveSemiSpace` images depending on whether or not you want VM assertion checking enabled. (A discussion of Jikes RVM configurations appears in Section 2.4.)

   Depending on your purposes you may want to choose another configuration that uses a different memory manager (See Section 8.1.2.).

   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 Jikes 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 Jikes RVM, and writes the Jikes RVM boot image. The boot image is the binary image of a ready-to-go instance of the Jikes 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 Jikes RVM Configurations

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

Most standard Jikes 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 Jikes 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

GenCopy a classic copying generational collector with a copying higher generation

GenMS a copying generational collector with a non-copying mark-and-sweep mature space

CopyMS a hybrid non-generational collector with a copying space (into which all allocation goes), and a non-copying space into which survivors go

WatsonSemispace the original "watson" copying semi-space collector

WatsonMarkSweep the original "watson" mark-and-sweep 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 Jikes RVM classes are included in the boot image (by default a small subset of these 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 9.

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 Jikes RVM build process consists of two major phases: the building of a *boot image*, and the building of a *boot loader*. The boot image is built using a Java program 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. To cross build, simply set your RVM_HOST_CONFIG and RVM_TARGET_CONFIG environment variables to be different files.

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_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*:

```
% 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 Jikes 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

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

variable shadowing by inner classes. We have successfully applied this patch to Jikes version 1.13 on both AIX©and Linux/IA32.

The `jBuildLibs` script will prompt you 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, this will *overwrite* the original version of `rvmrt.jar`, so be careful!

2.7 Building Documentation

The `jikesrvm-2.2.0.tar.gz` file contains a postscript version of this userguide in `$RVM_ROOT/rvm/doc`. Additionally, the developerWorks web page keeps an online 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.

3 Running Jikes RVM

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

3.1 Running Jikes RVM

Jikes 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 Standard 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 Jikes RVM.

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

`-D<name>=<value>` set a system property

`-verbose[:class|:gc|:jni]` enable verbose output

`-version` print current VM version and terminate

`-showversion` print current VM version and continue

`-fullversion` like version but with more information

`-?` or `-help` print help message

`-X` print help on non-standard options

`-jar` not supported

3.1.2 Non-standard Command-Line Options

The non-standard options are

`-X` print usage on nonstandard options

`-X:verbose` print out additional information for GC and hardware trap handling

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

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

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

-X:traceClassLoading produce a report on class loading activity. More details than -verbose:class.

-X:gc[:help] print options supported by memory management system

-X:aos:<option> pass <option> to the memory management system

-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

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

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

-X:base[:help] print options supported by the baseline compiler when in a non-adaptive configuration

-X:base:<option> pass <option> to the baseline compiler when in a non-adaptive configuration

-X:opt[:help] print options supported by the optimizing compiler when in a non-adaptive configuration

-X:opt:<option> pass <option> to the optimizing compiler when in a non-adaptive configuration

-X:prof:<option> pass <option> to the profiling subsystem

-X:vmClasses=<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:measureCompilation=<boolean> produce a report on compilation time

-X:measureClassLoading=<int> produce a report on class loading time (1 dump at end of run, 2 dump at every load).

-X:verify=<boolean> verify bytecodes

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

3.2 Regression Tests

The Jikes 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_ROOT/rvm/regression` subtree in the 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 `$RVM_ROOT/rvm/regression` directory and requires `ksh` and *GNU make* to run. When it is invoked, it will, by default, build a variety of boot images and run several test programs with each boot image. The behavior of RunSanityTests is highly customizable: most of its behavior can be controlled via command-line options, including what tests and boot images to run, whether to do sanity or performance tests, and what to do with test results. These options are described in more detail below, and a summary can be obtained by invoking `$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 Jikes 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 is 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 Jikes 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 Jikes 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 Jikes 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 Jikes 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 Jikes 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

```
gctest
bytecodeTests
opttests
reflect
threads
utf8
memberNaming
jni
CaffeineMark
jBYTEmark
javalex
xerces
xalan
soot
SPECjvm98
SPECjbb2000
```

nightly-tests-short file

```
bytecodeTests
opttests
reflect
threads
CaffeineMark
jBYTEmark
javalex
```

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
BaseBaseGenCopy -test-list nightly-tests
BaseAdaptiveSemiSpace -test-list nightly-tests-short
FullOptMarkSweep -use-opt-levels "O0 O1 O2" -test-list nightly-tests
FullOptGenCopy -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 Overview of Jikes RVM Structure

This subsection briefly describes the overall structure of Jikes RVM. Details of the various subsystems are provided in subsequent sections.

4.1 Major Components of Jikes RVM

The Jikes RVM can be divided into the following major components:

Core runtime (thread scheduler, class loader, library support, verifier, etc.) This component is responsible for managing all the underlying data structures required to execute applications and interfacing with libraries.

Compilers (baseline, optimizing, JNI) This component is responsible for generating executable code from bytecodes.

Memory managers This component is responsible for the allocation and collection of objects during the execution of an application.

Adaptive optimization system This component is responsible for profiling an executing application and judiciously using the optimizing compiler to improve its performance.

More details of each of these components are provided in the following sections.

4.2 Package Structure

The 2.2.0 release introduced packages into the system. Prior releases did not use packages explicitly, which resulted in all classes being in the unnamed package.

There are currently seven packages in Jikes RVM. All classes are in one of these packages.

com.ibm.JikesRVM Classes for the core runtime, except for library support. This package also contains other classes that are not included in one of the other packages, such as the baseline and JNI compilers.

com.ibm.JikesRVM.librarySupport Classes for library support

com.ibm.JikesRVM.adaptive Classes for the adaptive optimization system

com.ibm.JikesRVM.memoryManagers.vmInterface Classes related to memory management that deal with the interface to the VM

com.ibm.JikesRVM.memoryManagers.JMTk Classes in the newer JMTk (Java Memory Management Toolkit) collection of memory managers

com.ibm.JikesRVM.memoryManagers.watson Classes in the older Watson collection of memory managers

com.ibm.JikesRVM.opt Classes related to the optimizing compiler, except for IR-related classes

com.ibm.JikesRVM.opt.ir Classes related to the IR (intermediate representation) of the optimizing compiler

The distributed directory structure does not follow the Java convention that the location of source file be correlated with the package structure, i.e., there is no `com/ibm/JikesRVM` directory anywhere under `rvmRoot`. Instead, a more logical structure of the source files is used. One step of building a boot image is copying the source files from the rvmRoot tree into a build directory.

The scripts that perform this copy create the directory structure required by Java semantics and place classes in appropriate directories.

This approach avoids the need to change the directory structure as the package structure evolves to become more fine-grained. If we eventually arrive at a more fine-grained package structure (so that each package contains a reasonably small number of classes) we may convert the source tree to a package-oriented structure to facilitate using standard Java IDE's to edit Jikes RVM source code.

5 Runtime Implementation Details

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

5.1 Object Model

An *object model* dictates how to represent objects in storage; the best object model will maximize efficiency of frequent language operations while minimizing storage overhead. Jikes RVM's object model is defined by `VM_ObjectModel.java`.

Overview Values in the Java programming language are either *primitive* (e.g. `int`, `double`, etc.) or they are *references* (that is, pointers) to objects. Objects are either *arrays* having elements or *scalar objects* having fields. Objects are logically composed of two primary sections: an object header (described in more detail below) and the object's instance fields (or array elements).

The Jikes RVM object model is governed by the following criteria:

instance 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,

other (less frequent) Java operations should not be prohibitively slow, and

per object storage overhead (ie object header size) should be as small as possible.

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) element of an array and the remaining elements are laid out in ascending order. The number of elements in an array, its *length*, is kept just before its first element. Thus, elements of the array can be easily accessed via base + scaled index addressing.

The Java programming language requires that an attempt to access an object through a `null` object reference generate a NullPointerException. In the Jikes RVM, references are machine addresses, and `null` is represented by address 0. The AIX$^{\mathrm{TM}}$[7] 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.[8] 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.

Object Header Logically, every object header contains the following components:

TIB Pointer The TIB (Type Information Block) holds information that applies to all objects of a type. The structure of the TIB is defined by `VM_TIBLayoutConstants.java`. A TIB includes the virtual method table, a pointer to an object representing the type, and pointers to a few data structures to facilitate efficient interface invocation and dynamic type checking.

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

[8] 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.)

Default Hash Code Each Java object has a default hash code.

Lock Each Java object has an associated lock state. This could be a pointer to a lock object or a direct representation of the lock.

Array Length Every array object provides a *length* field that contains the length (number of elements) of the array.

Garbage Collection Information Each Java object has associated information used by the memory management system. Usually this consists of one or two mark bits, but this could also include some combination of a reference count, forwarding pointer, etc.

Misc Fields In experimental configurations, the object header can be expanded to add additional fields to every object, typically to support profiling.

An implementation of this abstract header is defined by three files: `VM_JavaHeader.java`, which supports TIB access, default hash codes, and locking; `VM_AllocatorHeader.java`, which supports garbage collection information; and `VM_MiscHeader.java`, which supports adding additional fields to all objects.

As of Jikes RVM 2.1.0, there are four implementations of `VM_JavaHeader.java` that can be choosen at build time. Not all implementions will work with all memory managers! One ("default") implementation defines a two word java header and works with all memory managers. The other three define one word java headers. All three work with the markSweep allocator; two of the three (`oneWordMaskTIB` and `oneWordIndexTIB`) work with the semispace allocator. Currently none of the one word java headers have been tested with any other memory manager, although it should be fairly straightforward to get either `oneWordMaskTIB` or `oneWordIndexTIB` to work with any of the Jikes RVM memory managers.

5.2 Methods and Fields

A compiled method body is an array of machine instructions (stored as `ints`). 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 Jikes RVM's global data structures are stored in the JTOC. Literals, numeric constants and references to String constants, are also stored there. 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 `ints`, Jikes RVM uses a descriptor array, co-indexed with the JTOC, to identify the entries containing references. The JTOC is depicted in figure 1.

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 Jikes RVM's virtual method table. A virtual method dispatch entails loading the TIB pointer from the object reference, loading the address of the method body at a given offset off the TIB pointer, and making an indirect branch and link to it.

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.

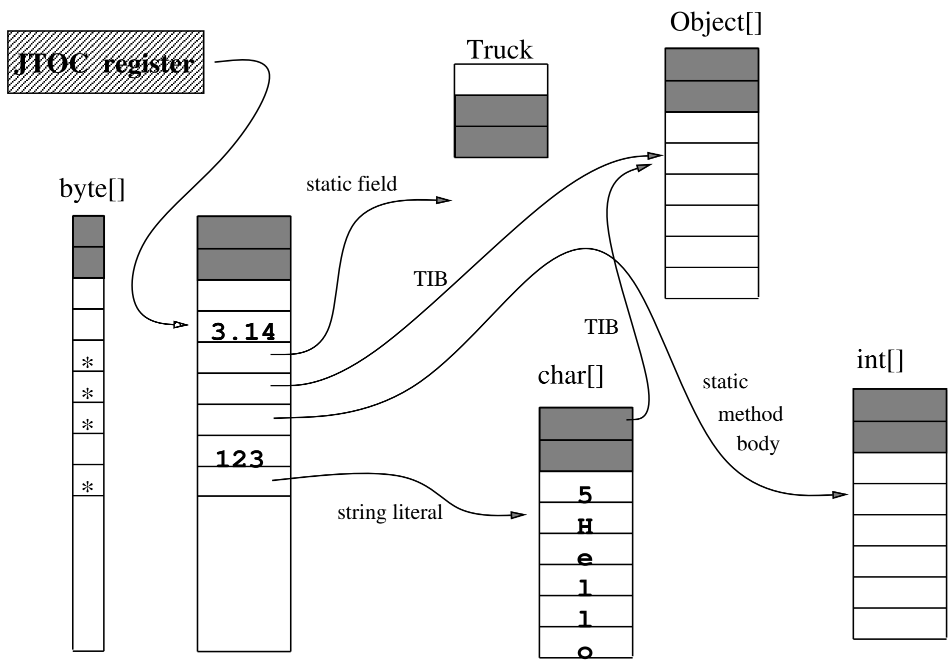

Figure 1: The Jikes RVM Table Of Contents and other objects.

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 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. For more details, see `VM_InterfaceInvocation.java`.

5.3 VM Conventions

5.3.1 AIX VM Conventions

This section describes register, stack, and calling conventions that apply to Jikes RVM on PowerPC™[9].

Stackframe layout and calling conventions may evolve as our understanding of the Jikes 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 into a global array of an object representing the current thread.
- **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 volatiles 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

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

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

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. Execute a stackoverflow check, and grow the thread stack if necessary.
2. Save the caller's next instruction pointer (callee's return address, from the Link Register).
3. Save any nonvolatile floating-point registers used by callee.
4. Save any nonvolatile general-purpose registers used by callee.
5. Store and update the frame pointer FP.
6. Store callee's compiled method ID
7. Check to see if the Java thread must yield the VM_Processor (and yield if threadswitch was requested).

Method epilogue responsibilities (some of these can be ommitted for leaf methods):

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.

5.3.2 Linux/IA32 VM Conventions

This section describes register, stack, and calling conventions that apply to Jikes 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 Nonvolatile.

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

EDI Nonvolatile. (used to hold JTOC in baseline compiled code)

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. The framePointer field of the VM_Processor object pointed to by ESI points to A's frame. 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.

5.4 Class Loading

Jikes 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.

5.5 Thread System

This section provides some explanation of how Java threads are scheduled and synchronized by the Jikes 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 Jikes RVM virtual processors to use is a command line argument (e.g. `-X:processors=4`) Generally, there should be one Jikes 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 `RVM_FOR_SINGLE_VIRTUAL_PROCESSOR` to 1. This may give a minor performance benefit on uniprocessors. See the configuration files `BaseBaseUPSemiSpace` and `OptOptUPSemiSpace` for examples of using this preprocessor directive.

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` does some additional housekeeping and transfers control to `dispatch` which selects the next thread to execute. Dispatch then invokes `VM_Magic.threadSwitch` to save the hardware context of A and restore the hardware context of B. It now appears as if B's previous call to `dispatch` has just returned and it continues executing. While dispatching is proceeding (from the time A is enqueued until B's hardware context is restored), the `beingDispatched` field of A is set to prevent it from being scheduled for execution on some other virtual processor while it is still executing in `morph` or `dispatch`.

Beginning with version 2.0.1, the Jikes RVM has a simple load balancing mechanism. Every once in a while, a thread will move from one virtual processor to the next. Such movement happens when a thread is interrupted by a timer tick (or garbage collection) or when it comes off a global queue (such as, the queues waiting for a heavy-weight lock, see `VM_Lock`). Such migration will be inhibited if the thread is the last (non-idle) executable thread on its current virtual processor.

If a virtual processor has no other executable thread, its idle thread runs. This thread posts a request for work and then busy-waits for a short time (currently 0.001 seconds). If no work arrives in that period, the virtual processor surrenders the rest of its time slice back to the operating system. If another virtual processor notices that this one needs work, it will tranfer an extra runnable thread (if it has one) to this processor. When work arrives, the idle thread yields to an idle queue, and the recently transferred thread begins execution.

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

If the Jikes RVM detects that a thread is stuck executing native code (JNI) for a long time, it temporarily prevents it from returning to Java and creates a new pThread (or recycles one previously created for this purpose) and transfers the stuck thread's virtual processor to the new pThread. When the stuck thread returns to Java, the pThread executing it is deactivated and added to a pool of available pThreads. Currently, this mechanism does not work on Linux and is disabled there.

Jikes RVM uses a light-weight locking scheme to implement Java monitors (see `VM_Lock` and `VM_ThinLock`). The exact details of the locking scheme are dependent on the which variant of `VM_JavaHeader.java` is selected at system build time. If an object instance has a light weight lock, then some bits in the object header are used for locking. If the top bit is set, the remainder of the bits 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.

5.6 VM Callbacks

The Jikes RVM provides callbacks for many runtime events of interest to the Jikes 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 5.4), 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 Jikes RVM supports callbacks for a number of events; see `VM_Callbacks` for the list of currently supported callbacks.

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()`).

Since the events for which callbacks are available are internal to the VM, 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 the `VM_GCStatistics` class.

5.6.1 An Example: Modifying SPECjvm98 to Report the end of a Run

The SPECjvm98 benchmark suite is configured to run one or more benchmarks a particular number of times. For example, the following runs the `compress` benchmark for 5 iterations:
`rvm SpecApplication -m5 -M5 -s100 -a _201_compress`

It is sometimes useful to have the vm notified when the application has completed an iteration of the benchmark. This can be performed by using the `VM_Callbacks` interface. The specifics are specified below:

1. Modify `spec/harness/ProgramRunner.java` as follows:
   (a) add an import statement for the `VM_Callbacks` class
   `import com.ibm.JikesRVM.VM_Callbacks;`
   (b) before the call to `runOnce` add the following:
   `VM_Callbacks.notifyAppRunStart(run);`
   (c) after the call to `runOnce` add the following:
   `VM_Callbacks.notifyAppRunComplete(run);`
2. Recompile the modified file using `javac` or `jikes`
   `javac -classpath .:$RVM_BUILD/RVM.classes:$RVM_BUILD/RVM.classes/rvmrt.jar spec/harness/ProgramRunner.java`
   or
   `jbuild.tool spec/harness/ProgramRunner.java`
3. Run Jikes RVM as you normally would using the SPECjvm98 benchmarks.

In the current system the `VM_Controller` class will gain control when these callbacks are made and print a message into the AOS log file (called AOSLog.txt, by default).

5.7 Support for Soot-style Annotations

The Jikes RVM optionally supports reading and using Soot-style class file annotations. Such annotations can specify that a null check or bounds check is redundant for a particular byte code instruction. These annotations are produced by the **Soot** class file optimizer available from http://www.sable.mcgill.ca/soot.

When the annotations command-line option is true, class file annotations are processed by `VM_Method.java` and stored internally in Jikes RVM. During compilation (either baseline or optimizing), the compiler queries the `VM_Method` class for a particular bytecode to determine if the generation of a null or bounds check can be suppressed.

Processing Soot-style annotations is not enabled, by default. To enable this support specify `''annotations=true''` to the appropriate compiler. For example, use `-X:irc:annotations=true` for non-adaptive images and `-X:aos:irc:annotations=true` for adaptive images.

6 Baseline Compiler Implementation Details

This section provides some implementation details for the baseline compiler.

6.1 General Architecture

The goal of the baseline compiler is to efficiently generate code that is "obviously correct." It also needs to be easy to port to a new platform and self contained (the entire baseline compiler must be included in all Jikes RVM bootimages to support dynamically loading other compilers).

Roughly 2/3 of the baseline compiler is machine-independent and located in the src/vm/compilers/baseline directory. The main file is `VM_BaselineCompiler`. The main platform-dependent file (located in src/vm/arch/{$arch}/compilers/baseline is `VM_Compiler`.

Baseline compilation consists of two main steps: GC map computation (discussed below) and code generation. Code generation is straightforward, consisting of a single pass through the bytecodes of the method being compiled. `VM_BaselineCompiler` contains the main code generation switch statement that invokes the appropriate `emit_<bytecode>` method of `VM_Compiler`.

6.2 GC Maps

The baseline compiler computes GC maps by abstractly interpreting the bytecodes to determine which expression stack slots and local variables contain references at the start of each bytecode. There are additional compilations to handle `JSR`s; see the source code for details. This strategy of computing a single GC map that applies to all the internal GC points for each bytecode slightly constrains code generation. The code generator must ensure that the GC map remains valid at all GC points (including implicit GC points introduced by null pointer exceptions). It also forces the baseline compiler to report reference parameters for the various `invoke` bytecodes as live in the GC map for the call (because the GC map also needs to cover the various internal GC points that happen before the call is actually performed). Note that this is not an issue for the optimizing compiler which computes GC maps for each machine code instruction that is a GC point.

6.3 Command-Line Options

The command-line options to the baseline compiler are stored as fields in an object of type `VM_BaselineOptions`; this file is mechanically generated by the build process.

To add or modify the command-line options in `VM_BaselineOptions.java`, you must modify either `BooleanOptions.dat`, or `ValueOptions.dat`. You should describe your desired command-line option in a format described below in Section 7.1, which also provides the details for the optimizing compiler's command-line options. Some options are common to both the baseline compiler and optimizing compiler. They are defined by the `SharedBoolanOptions.dat` and `SharedValueOptions.dat` files found in the rvm/src/vm/compilers/utility directory.

7 Optimizing Compiler Implementation Details

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

7.1 Options

The Jikes RVM stores command-line options to the optimizing compiler as fields in an object of type `OPT_Options`. The Jikes 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.

7.1.1 BooleanOptions.dat

The `BooleanOptions.dat` file defines boolean options for the optimizing compiler. The file describes each command-line option by a two-line record, and each record is separated by a blank line. Long lines can be continued 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 hold a short textual description of the semantics of the option. The system will print this description when so instructed by the command-line option `-X:irc:help` (see Appendix A).

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
```

7.1.2 ValueOptions.dat

The `ValueOptions.dat` file defines non-boolean options for the optimizing compiler. The file describes each command-line option with a three-line record, and each record is separated by a blank line. As with `BooleanOptions.dat`, long lines can be continued 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 hold a short textual description of the semantics of the option. The system will print this description when so instructed by the command-line option `-X:irc:help` (see Section A).

The third line of each record is used for enumeration options, and must be left blank for non-enumeration 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 must 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, Jikes 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, Jikes 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()`.

7.2 Method Compilation

The fundamental unit for optimization in the Jikes 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. An instance of this class includes, among other fields, the `VM_Method` to be compiled and the array of `OPT_OptimizationPlanElements` which define the compilation steps. The `execute` method of an `OPT_CompilationPlan` invokes the optimizing compiler to generate machine code for the method, executing the compiler phases as listed in the plan's `OPT_OptimizationPlanElements`.

The `OPT_OptimizationPlanner` class defines the standard phases used in a compilation. This class contains a static field, called `masterPlan`, which contains all possible `OPT_OptimizationPlanElements`. 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 client (compiler driver) constructs a specific optimization plan by including all the `OPT_OptimizationPlanElements` 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 two optimization plans associated with different methods will share the same component element objects. Clearly, it is undesirable to share state associated with a particular compilation phase between two different method compilations. 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 [10].

7.3 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™[11].

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 7.4 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.

[10] Since this coding pattern may result in unintended memory leaks, overriding `newExecution` may be disallowed in future releases.

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

7.4 Instruction Formats

Every IR instruction fits one of the pre-defined *Instruction Formats*. The Java package `instructionFormats` 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:

```
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.

7.5 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™[12]. 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.

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

- 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.

- 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`.

7.6 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 `BaseBaseOTHSemiSpace`; this image provides a BaseBase configuration with 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.

```
% 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 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`.

8 Memory Management Details

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

Version 2.2.0 of Jikes RVM introduces JMTk, a new family of memory managers. JMTk is described in the next section. A few of the original watson collectors are still included in Jikes RVM 2.2.0 but are scheduled for removal in the 2.2.1 release. Only the Semispace and MarkSweep watson collectors have been regularly tested.

8.1 JMTk

This section provides information on the implementation of JMTk, the memory management component of Jikes RVM runtime system.

From Jikes RVM 2.2.0 onward, JMTk (Java Memory management Toolkit) became the default memory management system for Jikes RVM. JMTk is designed to be a portable toolkit, with Jikes RVM-specific code factored out as far as possible.

8.1.1 Directory Structure and Packages

JMTk classes are contained in the `$RVM_ROOT/rvm/src/vm/memoryManagers/JMTk` directory.

In keeping with the goal of modularity and portability, as far as possible Jikes RVM-specific code is factored out. Such VM-specific code resides in a separate package, (`com.ibm.JikesRVM.memoryManagers.vmInterface`), and can be found in the `vmInterface` subdirectory. All other code is part of the `com.ibm.JikesRVM.memoryManagers.JMTk` package.

The `plan` sub-directory contains classes that define *memory management plans*. A plan specifies a particular configuration of JMTk components which together define the memory management strategy for a particular build of Jikes RVM. The `policy` sub-directory contains classes implementing various memory management policies (such as mark-sweep collection, free-list allocation, bump-pointer allocation, etc.). The `utility` sub-directory contains classes implementing generic utilities (such as load-balancing parallel queues, sequential store buffers, etc.).

8.1.2 Choosing a Garbage Collector

Depending on your purposes, you may choose to build Jikes RVM with one of the following plans:

- `SemiSpace` (copying),
- `MarkSweep` (non-copying),
- `GenCopy` (classic copying generational),
- `GenMS` (generational with mark-sweep mature space), or
- `CopyMS` (non-generational copy/mark-sweep hybrid)
- `NoGC` (allocation only, no garbage collection)

The relative performance of these collectors is highly dependent on the application. `GenMS` is probably a good starting point as it has good performance in a wide range of situations.

All of the memory managers support finalization. Moreover, a collection can proceed even when some threads are executing in native code. When a collection starts, threads in native code are blocked from returning to Java code for the duration of that collection.

8.1.3 Plans

All plans inherit from `BasePlan`, and all plans in this release are "stop the world" collectors, so they all inherit from `StopTheWorldGC`, which implements basic stop the world GC functionality. The two generational collectors both inherit from `Generational`, which includes common nursery and write barrier implementations.

All JMTk plans support parallel allocation and collection. To minimize synchronization overheads, unsynchronized *thread-local* actions are distinguished from *global* actions, which must only be performed by a single thread. Global state is held in each plan's class variables, while instance variables reflect thread-local state, each `Plan` instance bound to a `VM_Processor` instance.

The basic functions of each plan include:

- Identifying a virtual memory layout (using `VMResource` to, for example, bind a semi-space or the nursery to a particular address range).
- Providing allocation by binding suitable allocators to different `VMResource`s.
- Invoking collection when necessary through the use of a *polling* mechanism.
- Applying the appropriate collection policies to objects encountered during the collection process (objects may be subject to different collection regimens depending on where they reside in memory).
- Implementing read and write barriers if necessary.

It is not difficult to add your own memory management plan (allocator and collector) to JMTk, especially if it uses the same "stop the world" parallel collection strategy used by all the collectors in this release. A good way to start is to compare some of the different plans and understand the significance of the differences.

The basic steps are:

1. Create a new directory within the plan subdirectory, such as "NewGC".
2. Add a new configuration in `$RVM_ROOT/rvm/config/build` which includes your new directory in the build. Name it appropriately, such as "BaseBaseNewGC".
3. Modify `$RVM_ROOT/rvm/bin/jconfigure` to handle the new collector sub-directory.
4. Copy the contents of some existing plan directory into your new directory, and modify the files (`Plan.java` and `Header.java`), choosing a starting point that has similar properties, such as copying or non-copying, generational or non-generational.

8.1.4 Policy

The `policy` sub-directory contains implementations of key memory management policy choices, such as bump-pointer allocation and free-list allocation, copying collection and mark-sweep collection.

8.1.5 Utility

The `utility` sub-directory contains basic utilities and mechanisms, including:

- Load-balancing shared parallel queues, which provide load balancing queue management for address queues (used by the GC work queue), and sequential store buffers (used by some write barriers as a remembering mechanism).

- Memory resources, which are the mechanism for space *accounting*. Memory resources are used for accounting for all space, including space used by meta data (such as queues, etc.).
- Virtual Memory Resources (VM Resources), which are mechanism for virtual memory *mapping*. VM resources are used to associate regions of virtual memory with particular policies or needs (such as the nursery of a generational collector, or a region of memory where meta data resides, etc), and allow multiple allocators to consume each such space. There are a number of different VM resources, for monotonic allocation (used by bump pointer allocators), free list allocation (used by free list allocators), etc.
- Low level tools for `mmap`ping memory on demand, and for allocating raw memory (for use by meta data, for example).

8.1.6 VMInterface

The `vmInterface` sub-directory provides the interface between Jikes RVM and JMTk. The primary interface is provided by `VMInterface.java`. Key VM-dependent mechanisms include:

- Object, statics, and thread scanning,
- GC map iteration, and
- GC initiation and synchronization.

8.2 Deprecated Watson Collectors

8.2.1 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 original memory managers provided with the Jikes RVM and are described in this section.

In `watson`, the `common` subdirectory contains classes used by multiple memory managers, such as the load balancing work queue (see `VM_GCWorkQueue.java`) and write buffers `VM_WriteBuffer`. Other sub-directories of `watson` contain classes specific to a particular memory management strategy. Currently, there are 4 collectors supported: a semi-space copying collector (in `semispace`), a generational copying collector (in `copyGen`), a non-generational mark-sweep collector (in `markSweep`), and a generational hybrid collector in which the nursery is managed by copying collection and the older generation is managed by mark-sweep collection (in `hybrid`).

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

8.2.2 Choosing a Garbage Collector

Depending on your purposes, you may choose to build Jikes RVM with a semispace copying, mark-sweep noncopying, or hybrid collector. For programs that do not perform many collections, the copying collector will most likely perform best, since object allocation path lengths are shortest for this collector. On the other hand, a copying collector will generally require more memory than a mark-sweep collector to achieve the same overhead. For minimizing 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 non-copying. The best performance for a particular application should be determined by experiment.

All of the memory managers provided in the `watson` directory support finalization. Moreover, a collection can proceed even when some threads are executing in native code. When a collection starts, threads in native code are blocked from returning to Java code for the duration of that collection.

8.2.3 Writing A New Garbage Collector

It is not difficult to add your own memory manager (allocator and collector) to Jikes 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 collector, such as "NewGC".
2. Add a new configuration in `$RVM_ROOT/rvm/config/build` which includes your new directory in the build. Name it appropriately, such as "BaseBaseNewGC".
3. Modify `$RVM_ROOT/rvm/bin/jconfigure` to handle the new collector sub-directory.
4. 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, use `VM_Allocator.java` 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 Jikes RVM runtime (not all will be needed in any one implementation). A single class `VM_Allocator` implements both the methods that perform object allocation and the methods that perform garbage collections.

8.2.4 Load Balancing Work Queue

The class `VM_GCWorkQueue` implements a load balancing work-queue which is used by all the collectors to find all references reachable from some initial set of references. After the initial root scan, the work queue should hold all gray objects (*i.e.* objects that require scanning or further processing).

The work queue consists a pair of local buffers ("get" and "put") for each GC thread and a globally shared pool of buffers containing objects that need 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 buffer sizes (*e.g.* 256).

8.2.5 Generational Write Barrier

Jikes RVM provides a class `VM_WriteBarrier` to support generational garbage collection. When the static final field `VM_Allocator.writeBarrier` is true, the Jikes RVM compilers will generate write barriers by calling or inlining the methods in `VM_WriteBarrier`.

Currently, the generational collectors are the only ones requiring a write barrier and they all use the same one. Examination of `VM_WriteBarrier.java` shows that the write barrier consists of conditionally storing the address of the modified object into a per-processor sequential write buffer. For optimization, a *barrier bit* in the object header has been reserved to help eliminate unnecessary write buffer entries. When an object is allocated or processed during collection, the barrier bit is set. When an object is modified, the write barrier checks to see if the barrier bit is set. If so, the object is added into the write buffer entry and the barrier bit is cleared. Otherwise, the object has already been added to some write buffer and no action is required. It is important for a garbage collector to set the barrier bits so that subsequent stores into the object will not be lost.

The Jikes RVM generational collectors treat all objects which survive one garbage collection as old and performs *en masse* promotion. Thus, they do not need to maintain "remembered sets" of old objects to record intergenerational references between collections.

8.2.6 Starting Garbage Collection

Collector scheduling and suspension of mutator threads is handled by the classes `VM_Handshake` and `VM_CollectorThread`. A mutator thread initiates collection by calling the `collect` method of `VM_CollectorThread` which causes all collector threads to be scheduled on the `VM_Processor`'s, and then yields to allow the collector thread on the executing processor to be scheduled. During garbage collection, thread switching is disabled so the mutator does not execute until collection is completed. Parallel execution of collector threads is synchronized by barrier synchronization or 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 have arrived at the initial rendezvous. At all but the first rendezvous, it is known how many processors are expected to arrive at the barrier. Collection proper begins when 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).

Note that to implement a non-parallel collector, the configuration must define the variable `RVM_WITH_SINGLE_VIRTUAL_PROCESSOR`. Also, to implement a concurrent or incremental collector, it will be necessary to modify or extend the `VM_Handshake` and `VM_CollectorThread` classes.

8.2.7 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 VM exits. These include the number of collections, and the average and maximum collection times. For generational collectors, the times are grouped by minor and major collections. Some collectors provide additional information. For example, the copying collectors include the number of bytes copied. For more information, one can use the "`-verbose:gc=nnn`" where `nnn` indicates the level of verbosity. When no level is given, level 1 is assumed. Note that the more output there is, the more skewed the time measurements may become.

At verbosity level 2, the collector will show the time spent in each of the phases of garbage collection, such as stopping mutators, finding roots, marking reachable objects, and finalization. For each phase the time is shown at the end of each collection and the summary statistics shown when the VM exits. Since the cost of timing phases is minimal, it is always performed.

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.

MEASURE_WAIT_TIMES When this flag, which is in `VM_CollectorThread`, is enabled, the collector will measure the time each collector thread spends waiting during a collection. This includes 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 VM exits. If "`-verbose:gc`" is specified the output is generated after each collection.

COUNT_BY_TYPES and COUNT_BY_ALLOCATIONS These two flags control GC profiling and both flags have significant runtime cost. When `COUNT_BY_ALLOCATIONS` is enabled, the allocation code will track the number of bytes allocated, the number of objects allocated, and the number of objects that have lock fields. When `COUNT_BY_TYPES` is set, a group of per-type counters are enabled tracking both allocation and GC processing.

WORKQUEUE_COUNTS This flag is located in `VM_GCWorkQueue` and controls the counting of work queue buffers processed by each collector thread.

COUNT_GETS_AND_PUTS This flag is located in `VM_GCWorkQueue` and controls the counting of object references processed by each collector.

9 Adaptive Optimization System

A comprehensive discussion of the design and implementation of the adaptive optimization system is given in the OOPSLA 2000 paper by Arnold, Fink, Grove, Hind and Sweeney. Since the writing of the OOPSLA paper the following major changes have been made to the adaptive system:

- To improve the accuracy of the sampling mechanism, we insert yield points at method epilogues, as well method entries and loop backedges.
- The method samples are no longer decayed by the decay organizer. However, the call edge samples are still decayed.

The remainder of this section provides a road map to the key classes that comprise the adaptive optimization system. Most of the code for the adaptive system lives under `$RVM_ROOT/rvm/src/vm/adaptive`. One exception is the class `VM_RuntimeCompiler`, which lives in `$RVM_ROOT/rvm/src/vm/compilers/harness/runtime/adaptive`. This file contains the `compile` method that is called when a method needs to be *initially* compiled.

The files under `$RVM_ROOT/rvm/src/vm/adaptive` are organized into the following five directories:

controller: classes related to the controller thread, which is the decision-making component in the adaptive system

database: classes that contain information related to decisions made by the adaptive system, as well as cumulative profile data

recompilation: classes related to the recompilation thread

runtimeMeasurements: classes related to gathering online profile information

utility: miscellaneous adaptive system classes, such as options, logging facilities, and data structures.

Further details are provided in the following subsections.

9.1 Controller Directory

This directory contains classes related to the decision-making component of the adaptive system. Two key classes are `VM_Controller` and `VM_ControllerThread`. The former contains static data that is relevant to many components of the adaptive system. Some data items are references to controller, organizer, and (re-)compilation threads, the queues which these thread use to communicate, and command-line options. The boot method of this class boots the adaptive system and creates the `VM_ControllerThread` object.

The `VM_ControllerThread` creates the other adaptive system threads, related to profiling and recompilation. It then remains in a loop looking for events placed on the `controllerInputQueue` by the one or more organizers. The processing of these events is performed by other classes in this directory.

9.2 Database Directory

This directory contains classes that store information related to the adaptive system. We use the term "database" to mean an online repository, not a full-fledged database. This directory contains two subdirectories: `callGraph` and `methodSamples`, which contain information about a partial call graph (used by adaptive inlining) and a history of methods that are sampled. Both classes are populated with data based on the yield-point sampling mechanism.

9.3 Recompilation Directory

This directory contains the VM_CompilationThread, which processes events placed on compilationQueue by the VM_ControllerThread, and invokes the optimizing compiler using the information passed in the event. Another class in this directory is VM_CompilerDNA, which contains the cost/benefit parameters that drive the VM_ControllerThread's recompilation decisions. This directory also contains a subdirectory called instrumentation, which contains classes related to inserting instrumentation into an optimized method. This enables the gathering of more detailed profile information, in addition to the yieldpoint-based profile.

9.4 RuntimeMeasurements Directory

In addition to two interfaces and a static class (VM_RuntimeMeasurements), this directory contains three subdirectories:

instrumentation: miscellaneous classes related to data collected from the instrumentation of methods.

listeners: classes that contain methods that are called when application samples, based on yield points, are taken. These samples occur when a Java thread calls into the Jikes RVM scheduler to perform a context switch to another Java thread. The sample is taken before the thread switch occurs. See the threadSwitch method in $RVM_ROOT/rvm/src/vm/scheduler/VM_Thread for details. There may be many active listeners at one time. For example, we may take regular method samples to determine recompilation candidates (VM_MethodListener) and call edge samples to aid in inlining decisions (VM_EdgeListener).

organizers: classes related to separate Java threads that process samples or other raw profile data. There may be many organizers active at the same time. For example, one organizer may process method samples (VM_MethodSampleOrganizer) while another is processing call edge samples (VM_AIByEdgeOrganizer).

9.5 Utility Directory

The adaptive system contains a logging mechanism that allows it to record actions it takes to a log file. The file VM_AOSLogging contains a series of methods that provide the ability to log decisions. The level of logging is controllable on the command line.

Option processing for the adaptive system is also found in this directory. It is similar to the option processing for the optimizing compiler. The command-line options to AOS are stored as fields in an object of type VM_AOSOptions; this file is mechanically generated 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 in Section 7.1. The options in the ShareBooleanOptions.dat and ShareValueOptions.dat files are defined as both AOS and optimizing compiler options.

10 Magic

This section provides information on "magic" which is an escape hatch that Jikes RVM provides to implement functionality that is not possible in pure Java . For example, the Jikes RVM garbage collectors and runtime system must, on occasion, access memory or perform unsafe casts. Users are *strongly* discouraged from using magic in their code except where absolutely necessary.

There are currently three types of magical operations which are described in the remainder of this section. The first is a collection of magical methods that are static methods of the class `VM_Magic`. The second is the magical class `VM_Address` used in parts of the runtime and garbage collector. The third is various mechanisms to declare code *uninterruptible*.

10.1 VM_Magic

Certain methods in the class `VM_Magic` are treated differently by the compiler. Because these methods access raw memory or other machine state, perform unsafe casts, or are operating system calls, they cannot be implemented in Java code. A Jikes RVM implementor 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 methods (see below) 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.

Since magic is inexpressible in Java, it is unsurprising that the bodies of `VM_Magic` methods are undefined. 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)[13]. Whenever the compiler encounters a call to one of these magic methods, it inlines appropriate code for the magic method into the caller method.

10.2 VM_Address

The type `VM_Address` is used to represent a machine-dependent address type. In the past, the base type `int` was used to represent addresses but this approach had several shortcomings. First, the lack of abstraction makes porting nightmarish. Equally important is that Java type `int` is signed whereas address are more appropriately considered unsigned. The difference is problematic since an unsigned comparison on `int` is inexpressible in Java.

To overcome these problems, instances of `VM_Address` are used to represent addresses. The class supports the expected well-typed methods like adding an integer offset to an address to obtain another address, computing the difference of two addresses, and comparing addresses. Other operations that make sense on `int` but not on addresses are excluded like multiplication of addresses. Two methods deserve special attention: converting an address into an integer and the inverse. These methods should be avoided where possible.

Without special intervention, using a Java object to represent an address would be at best abysmally inefficient. Instead, when the Jikes compiler encounters creation of an address object, it will return the primitive value that represents an address for that platform. Currently, the address type maps to a 32-bit unsigned integer. Since an address is not really an object, the following must be kept in mind:

1. Do not pass a `VM_Address` instance where an `Object` is expected. This will type-check but it not what you want. A corollary is to avoid overloading a method on both `Object` or a `VM_Address`.

[13] 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.

2. Do not synchronize on a `VM_Address` instance.

3. Due to a current shortcoming in the way `VM_Address` works, do not make an array of VM_Address values. This restriction may be removed in the future.

10.3 What are the Semantics of Uninterruptible Code?

Declaring a method uninterruptible enables a Jikes RVM developer to prevent the Jikes RVM compilers from inserting "hidden" thread switch points in the compiled code for the method. As a result, the code can be written assuming that it cannot involuntarily "lose control" while executing due to a timer-driven thread switch. In particular, neither yield points nor stack overflow checks will be generated for uninterruptible methods.

When writing uninterruptible code, the programmer is restricted to a subset of the Java language. The following are the restrictions on uninterruptible code.

1. Because a stack overflow check represents a potential yield point (if GC is triggered when the stack is grown), stack overflow checks are omitted from the prologues of uninterruptible code. As a result, all uninterruptible code must be able to execute in the stack space available to them when the first uninterruptible method on the call stack is invoked. This is typically about 8K for uninterruptible regions called from mutator code. The collector threads must preallocate enough stack space, since all collector code is uninterruptible. As a result, using recursive methods in the GC subsystem is a bad idea.

2. Since no yield points are inserted in uninterruptible code, there will be no timer-driven thread switches while executing it. So, if possible, one should avoid "long running" uninterruptible methods outside of the GC subsystem.

3. Certain bytecodes are forbidden in uninterruptible code, because Jikes RVM cannot implement them in a manner that ensures uninterruptibility. The forbidden bytecodes are: aastore, invokeinterface, new, newarray, anewarray, athrow, checkcast and instanceof unless the LHS type is a final class, monitorenter, monitorexit, mulitanewarray.

4. Uninterruptible code cannot cause class loading and this must not contain unresolved getstatic, putstatic, getfield, putfield, invokevirtual, or invokestatic bytecodes.

5. Uninterruptible code cannot contain calls to interruptible code. As a consequence, it is illegal to override an uninterruptible virtual method with an interruptible method.

6. Uninterruptible methods cannot be synchronized.

We have augmented the baseline compiler to print a warning message when one of these restrictions is violated. Because there are still a small number of violations in Jikes RVM, this checking is not enabled by default, but can be enabled by setting `VM_Configuration.VerifyUnint` to true. If uninterruptible code were to raise a runtime exception such as NullPointerException, ArrayIndexOutOfBoundsException, or ClassCastException, then it could be interrupted. We assume that such conditions are a programming error and do not flag bytecodes that might result in one of these exceptions being raised as a violation of uninterruptibility. Checking for a particular method can be disabled by having the method throw the exception `VM_PragmaLogicallyUninterruptible`. This should be done with extreme care, but in a few cases is necessary to avoid spurious warning messages.

The following rules determine whether or not a method is uninterruptible.

1. All class initializers are interruptible, since they can only be invoked during class loading.

2. All object constructors are interruptible, since they an only be invoked as part of the implementation of the new bytecode.

3. If a method throws the exception `VM_PragmaInterruptible` then it is interruptible.
4. If none of the above rules apply and a method throws the exception VM_PragmaUninterruptible, then it is uninterruptible.
5. If none of the above rules apply and the declaring class of the method directly implements the interface `VM_Uninterruptible` then it is uninterruptible.

Whether to use `VM_Uninterruptible` or the `VM_PragmaUninterruptible` is a matter of taste and mainly depends on the ratio of interruptible to uninterruptible methods in a class. If most methods of the class should be uninterruptible, then implementing `VM_Uninterruptible` is preferred.

11 JNI Implementation Details

This section describes how Jikes RVM interfaces to native code. There are three major aspects of this support:

JNI Functions This is the mechanism for transitioning from native code into Java. Jikes RVM implements all but 3 of the functions defined by the 1.1 JNI specification. Jikes RVM implements almost none of the additional functions added by the 1.2 JNI specification.

Native methods This is the mechanism for transitioning from Java to native code. In addition to the normal mechanism used to invoke a native method, Jikes RVM also supports a more restricted syscall mechanism that is used internally by low-level VM code to invoke native code.

Integration with n-to-m threading Attempting to get Jikes RVM's cooperative n-to-m threading model to work nicely (at all) with native code is a major challenge. We have gone through several major redesigns of the JNI support code and RVM thread system in the process. This is still a work in progress.

Each of these aspects is discussed in more detail in the following sections.

11.1 JNI Functions

All but three of functions defined by the 1.1 JNI specification are implemented by Jikes RVM. The missing functions are `DefineClass`, `RegisterNatives`, and `UnregisterNatives`. Almost all of the additional functions added by the 1.2 JNI specification are not implemented. Contributions of the missing functionality (and associated tests) would be greatly appreciated.

The functions are defined in the class `VM_JNIFunctions`. Methods of this class are compiled with special prologues/epilogues that translate from native calling conventions to Java calling conventions and handle other details of the transition related to m-to-n threading. Currently the optimizing compiler does not support these specialized prologue/epilogue sequences so all methods in this class are baseline compiled. The prologue/epilogue sequences are actually generated by the platform-specific `VM_JNICompiler`.

11.2 Invoking Native Methods

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 a native method by using the platform-specific `VM_JNICompiler` to generate a stub routine 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.

11.3 Interactions with m-to-n Threading

See section 5.5 for more details on the thread system and m-to-n threading in Jikes RVM.

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. Note that this preemption mechanism is currently only implemented on AIX.

On Linux/IA32 Jikes RVM "hijacks" certain blocking system calls and reflects them back into the VM. The VM then uses nonblocking equivalents. This handles many of the common cases of blocking native code without requiring the full complexity of the timer-based preemption mechanism used on AIX. A complete solution would consist of implementing both mechanisms on both platforms. We hope to do this in the future.

11.4 Implementation Details

Supporting the combination of blocking native code and m-to-n threading is inherently complicated. Unfortunately the Jikes RVM implementation is further complicated by the fact that too much of the control logic for Java-to-C and C-to-Java transitions is embedded in the low-level, platform-specific VM_JNICompiler classes. As a result the code is very hard to maintain and the JNI implementations on different platforms have an unfortunate tendency to diverge.

We have some ideas for a redesign that would enable more of the control logic to be embodied in shared Java code, but there are a few minor issues to be worked out. Hopefully this will happen eventually.

12 Class Libraries

As of version 2.2.0, Jikes^{TM}[14] RVM has mostly switched to using the GNU Classpath class libraries. The default library configuration is a "hybrid" configuration that uses the original OTI-derived class libraries for the `java.lang`, `java.lang.ref`, `java.lang.reflect`, `java.util`, `java.io`, and `java.net` packages. All other packages come from GNU Classpath.

There is also an experimental "pure" configuration that uses only the GNU Classpath class libraries. Although we are moving towards making the classpath-only configuration the default for Jikes RVM, it is not yet ready for general use. We hope to make this the default in the near future.

Some library classes require special VM support. This is provided by `com.ibm.JikesRVM.librarySupport` and the classes found in `rvm/src/vm/OTILibrarySupport` and `rvm/src/vm/GNUClasspathSupport`.

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

13 Debugging Support

Previous releases of Jikes^{TM}[15] RVM included a home grown debugger called jdp. The jdp debugger provided some useful functionality for debugging Jikes RVM itself, but has always been a little unstable and was not really intended to support source level debugging for programs running on top of Jikes RVM. Prior to the 2.2.0 release, the Jikes RVM core team decided to deprecate jdp because we believe that the cost of maintaining it outweighs its usefulness.

The future plan for debugging Jikes RVM consists of two main pieces of work.

- Implement JDWP support in Jikes RVM. If Jikes RVM implemented the standard debugging wire protocol (JDWP), then a number of debugger front ends could be used with Jikes RVM. This would support most debugging tasks, but would obviously not work if Jikes RVM itself was crashing or corrupted.
- Low level debugging of Jikes RVM itself would be performed using gdb. Work could be done either in gdb or Jikes RVM to make this marginally more pleasant. For example, Jikes RVM could be extended to generate basic stab information for the bootimage.

Contributions from the community along either of these fronts would be greatly appreciated. Some prototyping of JDWP support for Jikes RVM was done at IBM during the summer of 2002; we can make this code available if anyone is interested in completing it.

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

14 Using the Jikes RVM to Profile an Application

This section contains information on several ways in which Jikes RVM can be used to profile an application and the VM itself. The first section describes how Jikes RVM supports platform-specific hardware performance monitors (HPM). Currently HPM support is only available for PowerPC$^{\mathrm{TM}}$[16]/AIX$^{\mathrm{TM}}$[17] version 5, but can be used in any configuration of Jikes RVM on that platform. The next two sections describe how adaptive configurations of Jikes RVM can be used to gather profile data. This support is available on all platforms, but only in adaptive configurations.

14.1 Using Hardware Performance Monitors

Jikes RVM can be built to enable access to the PowerPC hardware performance monitors (HPM) using the `bos.pmapi` interface included in AIX$^{\mathrm{TM}}$[18] version 5. To build Jikes RVM to use HPMs, you must define RVM_HPM_DIR to be the directory containing the pmapi code (typically /usr/pmapi) and RVM_WITH_HPM to be 1. See the config file $RVM_ROOT/rvm/config/powerpc-ibm-aix5.1.MC4U for an example.

After a valid configuration is built, the HPM are accessed through command line options. There are two options available:

- `-X:hpm:event<k>=N` where `k` is an integer (starting from 1) that identifies an HPM counter, and `N` is an integer that identifies, for the identified counter, the event that is counted. The number of counters and the events associated with a specific counter are microprocessor specific. For example, the PowerPC 604e only has four counters where the PowerPC 630 and Power 4 have eight. The PowerPC 604e reserves event number 2 on every counter to count the number of instructions completed whereas the PowerPC 630 reserves event number 3.
- `-X:hpm:mode=N` controls when events are counted: `N` = 4 when user code is running, `N` = 8 when kernel code is running, or `N` = 12 when both user and kernel code is running.

At least one HPM event command line option must be set for HPM counter values to be gathered. HPM only works accurately when the number of virtual processors is one, `-X:processors=1`. The output for a PowerPC 604e machine when the command line arguments are

```
% rvm -X:hpm:event1=6 -X:hpm:event2=6 -X:hpm:event3=2 -X:hpm:event4=1 -X:hpm:mode=12
```

looks like:

```
*** Configuration :
Mode = kernel and user; Process tree = off; Thresholding = off
Counter  1, event  6: PM_DTLB_MISS
Counter  2, event  6: PM_DC_MISS
Counter  3, event  2: PM_INST_CMPL
Counter  4, event  1: PM_CYC

*** Results :
PMC 1    PMC 2    PMC 3      PMC 4
=====    =====    =====      =====
4300056  8018925  1002604892  2412445847
```

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

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

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

14.2 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 Jikes RVM to gather profile data.

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

14.3 Instrumented Event Counters

This section describes how the Jikes 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 14.3.1 describes existing instrumentation phases and how to run them, and Section 14.3.2 describes the details of how a new phase can be added.

14.3.1 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 14.3.2), 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_counters=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`

14.3.2 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.

15 Experimental Guidelines

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

15.1 Which boot image should I use?

To make a long story short the best performing configuration of Jikes RVM will almost always be `FastAdaptive`*¡some GC¿*. Unless you really know what you are doing, don't use any other configuration to do a performance evaluation of Jikes RVM. In most cases, `FastAdaptiveSemiSpace` is likely to be your best choice. You can use the `FastSemiSpace` configuration if you want to force all dynamically loaded methods to be compiled by the optimizing compiler, but this configuration will have extremely poor startup behavior and will not perform any profile-directed optimizations.

Any bootimage you use for performance evaluation must have the following characteristics for the results to be meaningful:

- `RVM_WITHOUT_ASSERTIONS=1`. Unless this is set, the runtime system and optimizing compiler will perform fairly extensive assertion checking. This introduces significant runtime overhead. By convention, a configuration with the `Fast` prefix disables assertion checking.
- `RVM_WITH_OPT_BOOTIMAGE_COMPILER=1`. Unless this is set, the bootimage will be compiled with the baseline compiler and virtual machine performance will be abysmal. Jikes RVM has been designed under the assumption that aggressive inlining and optimization will be applied to the VM source code.
- Any configuration that performs opt compilation at runtime (`RVM_WITH_ADAPTIVE_SYSTEM=1` or `RVM_WITH_OPT_RUNTIME_COMPILER=1`) should be built with `RVM_WITH_ALL_CLASSES=1`. This includes the optimizing compiler and associated support classes in the bootimage where they can be optimized by the bootimage compiler. By convention, configurations that include the opt compiler in the bootimage have the `Full` or `Fast` prefix. Configurations where `RVM_WITH_ALL_CLASSES` is not set to 1 that use the optimizing compiler will dynamically load it (which will force it to be baseline compiled, *even if RVM_WITH_OPT_RUNTIME_COMPILER is set to 1*).

15.2 What command-line arguments should I use?

For best performance we recommend the following:

- `-processors all`: By default, Jikes 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 Jikes RVM thread and synchronization implementation do not play well with others.

15.3 Jikes RVM is really slow! What am I doing wrong?

Perhaps you are not seeing stellar Jikes RVM performance. If Jikes RVM as described above is not competitive with the IBM AIX™[19] or Linux/IA32 product DK, we recommend you test

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

your installation with the SPECjvm98 benchmarks. We expect Jikes RVM performance to be competitive with the IBM DK 1.3.0 on the SPECjvm98 benchmarks.

Of course, SPECjvm98 does not guarantee that Jikes 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 somewhat less mature than the PPC port, and does not deliver competitive performance on some codes. In particular, IA32 floating-point performance is mediocre.

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

- Jikes 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 Jikes RVM to perform well on codes with many threads, such as VolanoMark. However, if you have a code with many threads, each using JNI, Jikes RVM performance will suffer due to factors in the design of the current thread system.
- Jikes 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 Jikes 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 Jikes RVM virtual processor (ie., pthread) has no work to do, it spins chewing up cpu cycles. Thus, Jikes 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 adaptive system may not perform well on SMPs; this may be due to bad interaction with the thread load balancer.

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

16 Coding Style Guidelines

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

Regrettably, much code in the current system does not follow any consistent coding style. This is an unfortunate residue of the system's evolution make editing sometimes unpleasant, and prevents 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.

16.1 Coding style description

The Jikes 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 space indenting and allow 132 column lines.

16.2 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.

16.3 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.

16.3.1 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)
```

16.3.2 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 (16.3.2).

vim If you are using vim version 6.0 or higher, then you can 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 formatoptions+=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
```

17 FAQ

Jikes RVM Frequently Asked Questions

17.1 General

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

17.1.1 What is Jikes 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 these platforms. The Jikes RVM includes the latest VM technologies for dynamic compilation, adaptive optimization, garbage collection, thread scheduling, and synchronization.

17.1.2 Who is using the Jikes RVM

A list of current Jikes 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.

17.1.3 Can I use Jikes RVM when teaching a class?

Yes, this is fine under the Common Public License. In fact, professors have already done at both the graduate and undergraduate levels. Teaching material and links to courses taught using Jikes RVM are available at `http://www.ibm.com/developerworks/oss/jikesrvm/info/course-info.shtml`.

17.1.4 Who can I contact with questions?

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

17.1.5 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.

[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.

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

17.1.6 How can I contribute to RVM?

Bug reports or feature requests can be submitted directly at `http://www-124.ibm.com/developerworks/projects/jikesrvm`. Information about contributing bug fixes or extensions to the system can be found at `http://www.ibm.com/developerworks/oss/jikesrvm/info/contributions.shtml`.

17.2 Getting RVM and Documentation

17.2.1 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.

17.2.2 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.

17.2.3 Is there documentation on-line?

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

17.2.4 Can I get the Quicksilver Quasi-Static System?

No. This project is no longer active or supported.

17.2.5 Can I get DejaVu?

No. This project is not supported.

17.2.6 What happened to `jdp`?

The jdp debugger has been deprecated. See Section 13 for more details.

17.3 Building RVM

17.3.1 What's the right value to use for RVM_FOR_SINGLE_VIRTUAL_PROCESSOR?

Jikes RVM can be built to support either m-to-n or m-to-1 threading. In m-to-n threading (RVM_FOR_SINGLE_VIRTUAL_PROCESSOR equals 0), the virtual machine multiplexes m Java onto n virtual processors (operating system pthreads). In m-to-1 threading (RVM_FOR_SINGLE_VIRTUAL_PROCESSOR equals 1), all m Java threads run on a single virtual processor. Jikes RVM can only exploit multiple CPUs on a hardware SMP machine if built with RVM_FOR_SINGLE_VIRTUAL_PROCESSOR equal to 0.

Unfortunately, m-to-n threading is not supported on all platforms due to mismatches between Jikes RVM and the host pthread implementation. The key issue is that because Jikes RVM multiplexes Java threads (each with their own stack) onto operating system threads, the C libraries it is linked with must not use the value of a thread's stackpointer/framepointer to access pthread-local storage. The preferred values for RVM_FOR_SINGLE_VIRTUAL_PROCESSOR are as follows:

- `AIX/PowerPC`: 0 (SMP is supported)
- `Linux/PowerPC`: 1 (SMP is not supported)
- `Linux 2.2/IA32`: 1 (SMP is not supported)
- `Linux 2.4/IA32`: 0 (SMP is supported), but with the additional restriction that you must link with a version of glibc that was compiled to use the GS segment register to access pthread-specific state. Depending on your Linux distribution, this may not be the default for glibc. See a discussion in the archives for jikesrvm-researchers from November of 2001. If your version of glibc is not compiled this way, Jikes RVM will usually fail by 'hanging' when it is run.

17.3.2 Which jikes should I use?

At Watson, we're currently using `jikes` v1.18 to compile the RVM source on both Linux and AIX™[23]. Previously we were using version 1.13. Version 1.14 through 1.17 have been reported to not entirely work with Jikes RVM. In order to build the `rvmrt.jar` library with jikes 1.13 , we applied patch 62 to the jikes build to fix a jikes scoping problem. For versions of Jikes RVM before 2.2.0, you must use jikes 1.13.

17.3.3 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.

17.3.4 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.

17.3.5 Can I monitor progress during jbuild?

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

17.3.6 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.

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

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`).

17.4 Runtime implementation

17.4.1 Does RVM have an interpreter?

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

17.4.2 Does RVM support JNI?

Most JNI functionality is supported. A few functions are not. There are a few functions that are only supported on AIX™[24]and some other functions that are only supported on Linux/IA32.

17.4.3 Does RVM support user-defined class loaders?

Not entirely. Jikes RVM version 2.1 implements many of the entrypoints for user-defined classloaders, so many codes should work. However, the underlying implementation does not yet support distinct namespaces. The net effect is that the RVM will not yet support two classes which have the same name, but different definitions due to different classloaders.

17.4.4 Does RVM support the Java security model?

We believe we support it to the extent that the GNU Classpath libraries support it.

17.4.5 Does RVM support serialization?

We believe we support it to the extent that the GNU Classpath libraries support it. We have noticed that sometimes it doesn't work exactly the same as in the Sun class library.

17.4.6 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.

17.4.7 Does RVM support zip files?

As of Jikes RVM v2.1.0, we use the GNU Classpath implementation of java.util.zip and java.util.jar.

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

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

17.4.8 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.

On some platforms, only m-to-1 threading is supported. See question 17.3.1.

17.4.9 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.

17.4.10 How can I implement a new GC algorithm?

See the memory management section of the userguide.

17.4.11 How does RVM enter native code?

See the discussion in section 11.

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

See the discussion in section 11.

17.4.13 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 to implement attempt; prepare is a normal load instruction.

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.

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

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

17.5 Libraries

17.5.1 Does RVM run awt?

We believe we support it to the extent that the GNU Classpath libraries support it. The last time we tried it, classpath support was very incomplete.

17.5.2 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.

17.5.3 Why don't you use the GNU Classpath libraries?

As of version 2.2.0 we mostly do. We plan to switch completely to the GNU Classpath libraries in the near future. See section 12 for more details.

17.6 Optimizing Compiler

17.6.1 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.

17.6.2 What is AOS?

AOS stands for adaptive optimization system.

17.6.3 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.

17.6.4 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.

17.6.5 What is Heap Array SSA form?

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

17.6.6 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.

17.6.7 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`.

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

See section 7.2 of the userguide, which describes how to add phases to class `OPT_OptimizationPlanner`.

17.6.9 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 methods as they are invoked. 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.

[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.

17.6.10 What is the OptTestHarness?

The OptTestHarness (see section 7.6) 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.

17.7 Regression Tests

17.7.1 Where can I order or download the test programs SPECjvm98 and SPECjbb2000 ?

Information for ordering the SPEC benchmark suites can be found at http://www.spec.org. As you'll see from the site, there is a non-trivial charge for getting the suites, although universities do get a discount. However, because the license is institution-wide, you may want to check with other researchers at your institution to see if they already have a license.

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
- Perry Cheng
- Anthony Cocchi
- John Davis
- Julian Dolby
- Stephen Fink
- David Grove
- Mike Hind
- Matthias Hauswirth
- Wilson Hsieh
- Mark Mergen
- Igor Pechtchanski
- Barbara Ryder
- Stephen Smith
- Peter Sweeney
- Martin Trapp
- David Wong

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.

A Appendix A: Jikes RVM Command-Line Options

Currently, Jikes RVM has two types of configurations: the adaptive configurations that contain the adaptive optimization system (AOS), and the non-adaptive configurations that do not. This section describes the non-standard Jikes RVM command-line options that provide the mechanism to specify AOS, baseline compiler, and optimizing compiler options for the different types of configurations; and how to add a new non-standard command-line option.

A.1 Command-Line Options in Non-Adaptive Configurations

This section describes how non-standard Jikes RVM command line options are specified for a non-adaptive configuration of Jikes RVM. In a non-adaptive configuration, the command line options modify the behavior of the baseline or optimizing compiler. Different prefixes can be used to direct the option to one of following: the initial runtime compiler (`-X:irc:`), the baseline compiler (`-X:base:`) and the optimizing compiler (`-X:opt:`). Some prefixes are only allowed with some configurations. Depending on the configuration selected, the initial runtime compiler is either the baseline or the optimizing compiler.

Consider the case where the initial runtime compiler is the optimizing compiler. In that case either of the prefixes `-X:irc:` or `-X:opt:` can be used to pass options to the optimizing compiler. 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. In such a configuration the baseline compiler will still be present and will be used for some compilations. The prefix `-X:base:` must be used to pass an option to the baseline compiler in configurations when the initial runtime compiler is the optimizing compiler.

When the initial runtime compiler is the baseline compiler, the optimizing compiler will not be part of the system, so the prefix `-X:opt:` should not be used. Also, with such a configuration, options specified with the prefix `-X:irc:` must be options recognized by the baseline compiler.

All of the above non-standard VM options 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.

A.1.1 Optimizing Compiler Command-Line Options

To see descriptions of command-line options to the optimizing compiler, use the `help` option with the `-X:opt:` prefix to generate the following output:

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



Boolean Options (-X:opt:<option>=true or -X:opt:<option>=false)
Option                      OptLevel   Description
annotations                            Act on annotations in class files
preload_as_boot                        Apply boot options to preload_class
verbose                                Print method name at start of compilation
mc                                     Print final machine code
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
no_callee_exceptions                   Assert that any callee of this compiled method will not throw exceptions?
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                   0       Reorder basic blocks for improved locality and branch prediction
reorder_code_ph                1       Reorder basic blocks using Pettis and Hansen Algo2
```

inline_new	1	Inline allocation of scalars and arrays
inline	0	Inline statically resolvable calls
guarded_inline	1	Guarded inlining of non-final virtual calls
guarded_inline_interface	1	Speculatively inline non-final interface calls
static_splitting	1	CFG splitting to create hot traces based on static heuristics
redundant_branch_elimination	2	Eliminate redundant conditional branches
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?
live_range_splitting		Split live ranges using LIR SSA pass?
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
licm_ignore_pei		Asume PEIs do not throw or state is not observable
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?
code_patch_nop		Should all patch point be (unsafely) eliminated (at initial HIR)?
cw_referenced_classes		Apply Class Writer to referenced classes
cw_guards		Leave guards in before BURS
cw_stack		Process Stack to eliminate Loads/Stores
focusEffort		Focus compilation effort based on frequency profile data
phases		Print short message for each compilation phase
print_all_ir		Dump the IR after each compiler phase
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
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
print_cw_burs		Verbose dump of Class Writer BURS processing
print_cw_burs_noisy		Verbose dump of Class Writer BURS processing
print_cw_alloc		Verbose dump of Class Writer Allocation of Locals
print_cw_stack		Verbose dump of Class Writer Stack processing to eliminate Load/Store
print_cw_produce		Dump of Class Writer Byte Code Production
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)
print_mc_size		Report generated machine code size
skip_unknown_magic		Skip generation of unknown magic calls
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:opt<option>=<value>)

Option	Type	Description
preload_class	String	Class to preload upon 1st OPT compilation
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
infrequent_threshold	float	Cumulative threshold which defines the set of infrequent basic blocks
cbs_hotness	double	Threshold at which a conditional branch is considered to be skewed
ir_print_level	int	Only print IR compiled above this level

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

```
Selection of guard mechanism for inlined virtual calls that cannot be statically bound
inlining_guard              ig_method_test ig_class_test ig_code_patch
Selection of strictness level for floating point computations
fp_mode                     strict allow_fma allow_assoc
How to compute block and edge frequencies?
frequency_strategy          counters static dumb inverse
Selection of spilling heuristic
spill_cost_estimate         simple brainDead blockCount

Set Options (option is a set of values)
method_to_print              Only apply print options against methods whose name contains this string
exclude                      Exclude methods from being opt compiled
method_to_write              Write class file from HIR for this method
```

Note that when the initial runtime compiler is the optimizing compiler, the `-X:irc` and the `-X:opt` command line prefixes are equivalent.

A.1.2 Baseline Compiler Command-Line Options

To see descriptions of the command-line options to the baseline compiler, use the `help` option with the prefix `-X:base:` to generate the following output:

```
-X:base[:help] Print brief description of baseline compiler's command-line arguments
-X:base:printOptions Print the current values of the active baseline compiler options


Boolean Options (-X:base:<option>=true or -X:base:<option>=false)
Option                                 Description
annotations                            Act on annotations in class files
preload_as_boot                        Apply boot options to preload_class
verbose                                Print method name at start of compilation
mc                                     Print final machine code
edge_counters                          Insert edge counters on all bytecode-level conditional branches

Value Options (-X:base<option>=<value>)
Option                         Type    Description
preload_class                  String  Class to preload upon 1st OPT compilation
edge_counter_file              String  File into which to dump edge counter data

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

Set Options (option is a set of values)
method_to_print                Only apply print options against methods whose name contains this string
```

Note that when the initial runtime compiler is the baseline compiler, the `-X:irc` and the `-X:base` command line prefixes are equivalent.

A.1.3 Discussion

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

A.2 Command-Line Options in Adaptive Configurations

This section describes how non-standard Jikes RVM command line options are specified for an adaptive configuration. In an adaptive configuration, the command line options modifies the behavior of the adaptive optimization system, the optimizing compiler or the baseline 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 method 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, there is an initial runtime compiler. In the default adaptive configurations, the initial runtime compiler is the baseline compiler. Options are passed to the initial runtime compiler 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 A.1 for a discussion of the optimizing and baseline compiler command-line options that are available.

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 AOS and all optimization levels for method recompilation.

All of the above non-standard VM options 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.

A.2.1 Adaptive Optimization System (AOS) Command-Line Options

To see a description of the command-line options 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:irc[:help]    Print a brief description of intial runtime compiler's command-line options that initially compiles a method
-X:aos:base[:help]   Print a brief description of baseline compiler's command-line options that initially compiles a method
-X:aos:opt[:help]    Print a brief description of optimizing compiler's command-line options that recompiles 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?
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?
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?
early_exit                             Should AOS exit when the controller clock reaches EARLY_EXIT_VALUE?
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
lf                             String  Name of 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
decay_frequency                int     After how many clock ticks should we decay
decay_rate                     double  What factor should we decay 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
early_exit_time               int     Value of controller clock at which AOS should exit if EARLY_EXIT is true

Selection Options (set option to one of an enumeration of possible values)
Selection of primary controller strategy
primary_strategy             baseonly optonly adaptive

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. This has the effect of making the initial runtime compiler be the optimizing compiler (and no recompilations will take place), so subsequent options in the command line that are prefixed with `-X:aos:irc:` will be passed to the optimizing compiler. Since no recompilation will take place options prefixed with `-X:aos:opt` will have no effect. For example to optimize compile a method at optimization level 1, when the `optonly` strategy is specified, 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`.)

A.3 Adding Jikes RVM Non-Standard Command-Line Options

This section states how non-standard Jikes RVM command-line options can be added to Jikes RVM. Non-standard Jikes RVM command-line options are those options that are specific to Jikes RVM. The format of a non-standard Jikes 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.

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

B Appendix B: Jikes RVM Licenses

This section contains the licenses used for 1) the Jikes RVM, 2) the binary version of the Jikes RVM libraries, and 3) the source version of the Jikes RVM libraries. The GNU Classpath jazzlib license is available at `http://www.gnu.org/software/classpath`.

B.1 Jikes RVM License

The Jikes RVM is distribed under the Common Public License (CPL), which has been approved by the Open Source Initiative as a fully certified open source license.

THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and
b) in the case of each subsequent Contributor:
i) changes to the Program, and
ii) additions to the Program;
where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents " mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

2. GRANT OF RIGHTS

a) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.
b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition

of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.
c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.
d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

a) it complies with the terms and conditions of this Agreement; and
b) its license agreement:
i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;
ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;
iii) states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and
iv) states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.
When the Program is made available in source code form:

a) it must be made available under this Agreement; and
b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation is filed. In addition, If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. IBM is the initial

Agreement Steward. IBM may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

B.2 Jikes RVM Library (JLIBRARIES) Binary License

IBM JLIBRARIES

END USER LICENSE AGREEMENT

BEFORE DOWNLOADING, INSTALLING OR USING IBM JLIBRARIES, YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS.

DOWNLOADING, INSTALLING OR USING JLIBRARIES INDICATES YOUR ACCEPTANCE OF THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, YOU MAY NOT DOWNLOAD, INSTALL OR USE JLIBRARIES.

GRANT OF LICENSE:

(A) Definitions:

JIKES RVM is a research virtual machine.

JLIBRARIES means a set of libraries, in object code, which work with JIKES RVM. JLIBRARIES includes, without limitation, io, lang, net, and util.

(B) Licenses:

Subject to the terms and conditions of this Agreement, International Business Machines Corporation and its affiliates (collectively referred to in this Agreement as "IBM"), grant you a nontransferable, non-exclusive, internal use, limited copyright license to use

JLIBRARIES, in object code only, on each computer that you use. You may only use the JLIBRARIES with JIKES RVM.

Except for the limited rights granted above, IBM does not grant you any other rights with respect to JLIBRARIES including, without limitation, any right to distribute JLIBRARIES or any component of it, commercially or otherwise.

(C) Other License Information:

You may not reverse assemble, reverse compile, decompile, translate or otherwise attempt to discover the source code of any component of JLIBRARIES that is provided under this Agreement, except as permitted by law without the possibility of contractual waiver. In addition, you may not prepare derivative works of JLIBRARIES or any components of it. You may not alter any of the proprietary notices on JLIBRARIES.

You agree that any information, source code, and object code that you may provide to IBM related to JLIBRARIES is non-confidential and you grant to IBM, its affiliates, and its licensees a non-exclusive, fully-paid up, irrevocable license to use, make, have made, perform, prepare and have prepared derivative works, and sublicense such information, source code, and object code. You warrant that such information, source code, and object code is original to you and that you have the right to grant the licenses set out in this paragraph

LIMITATIONS: You acknowledge that under this Agreement you obtain only the rights to JLIBRARIES as expressly stated in this Agreement. No right, title or interest in or to any copyrights, trademarks, patents or other proprietary or intellectual rights relating to JLIBRARIES is transferred to you.

TERM AND TERMINATION: You may terminate your license to JLIBRARIES at any time by written notice to IBM and by destroying all your copies of JLIBRARIES. IBM may immediately terminate your license at any time if you fail to comply with the terms of this Agreement or if so required by a court of competent jurisdiction.

DISCLAIMER OF WARRANTY AND LIABILITY:

JLIBRARIES IS PROVIDED "AS IS" WITHOUT WARRANTY OR CONDITION OF ANY KIND EITHER EXPRESSED, IMPLIED, OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. THE ENTIRE RISK OF THE USE OR PERFORMANCE OF JLIBRARIES REMAINS WITH YOU. IN NO EVENT WILL IBM BE LIABLE FOR ANY LOST PROFITS, LOST SAVINGS, LOST REVENUE, LOST DATA OR FOR ANY DIRECT, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF IBM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES OR DAMAGES. IN ADDITION, IBM WILL NOT BE LIABLE FOR ANY DAMAGES CLAIMED BY YOU BASED ON ANY THIRD PARTY CLAIM.

Some jurisdictions do not allow the exclusion of implied warranties, or the limitation of consequential damages, so the above may not apply to you.

Note to U.S. Government Users - Documentation related to Restricted Rights - Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

Although IBM may try to answer technical support questions you have regarding your use of JLIBRARIES, any such assistance does not obligate IBM to provide support or maintenance services for JLIBRARIES.

GENERAL:

All terms and conditions, which by their nature should survive the termination or expiration of this Agreement, shall survive. If any provision of this Agreement is declared unenforceable at law, in whole or in part, the rest of the provisions of the Agreement shall remain in effect.

Only a written Agreement signed by both of us can modify this Agreement.

This Agreement is entered into in the State of New York, USA, and governed by the laws of the State of New York without regard to its conflict of law principles.

B.3 Jikes RVM Library (JLIBSOURCE) Source License

IBM JLIBSOURCE

END USER LICENSE AGREEMENT

IF YOU ARE A LICENSEE UNDER THE JLIBRARIES END USER LICENSE AGREEMENT (ABOVE), YOU MAY ALSO WISH TO BE LICENSED FOR RESTRICTED ACCESS TO THE SOURCE CODE ("JLIBSOURCE") FOR THE JLIBRARIES OBJECT CODE. IF SO, YOUR JLIBSOURCE LICENSE WILL BE GOVERNED BY THE TERMS EXPRESSED BELOW.

BEFORE DOWNLOADING, INSTALLING OR USING IBM JLIBSOURCE, YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS.

DOWNLOADING, INSTALLING OR USING JLIBSOURCE INDICATES YOUR ACCEPTANCE OF THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, YOU MAY NOT DOWNLOAD, INSTALL OR USE JLIBSOURCE.

GRANT OF LICENSE:

(A) Definitions:

JIKES RVM is a research virtual machine.

JLIBRARIES means a set of libraries, in object code, which work with JIKES RVM. JLIBRARIES includes, without limitation, io, lang, net, and util.

API's means the application programming interfaces for JLIBRARIES

JLIBSOURCE means the source code for JLIBRARIES .

(B) Licenses:

Subject to the terms and conditions of this Agreement, International Business Machines Corporation and its affiliates (collectively referred to in this Agreement as "IBM"), grant you a nontransferable, non-exclusive, internal use, limited copyright license:

i) to download and print JLIBSOURCE to enable you to view JLIBSOURCE to understand the functions of JLIBSOURCE for performance improvement and debugging purposes,

ii) for purposes of performance improvement and debugging only, to modify all parts of JLIBSOURCE that exclude API's and compile such modified JLIBSOURCE into a modified form of JLIBRARIES (that excludes API's), for your personal use only, on both these terms and the applicable terms of your JLIBRARIES license.

You may only use JLIBSOURCE, any modified JLIBSOURCE, and any modified JLIBRARIES with JIKES RVM. You may not modify API's.

Except for the limited rights granted above, IBM does not grant you any other rights with respect to JLIBSOURCE (or to any modified JLIBSOURCE or any modified JLIBRARIES) including, without limitation, any right to distribute JLIBSOURCE (or any modified JLIBSOURCE or any modified JLIBRARIES), commercially or otherwise.

(C) Other License Information:

Except as provided above, you may not prepare derivative works of JLIBSOURCE. You may not alter any of the proprietary notices on JLIBRARIES.

You agree to supply to IBM, in a form acceptable to IBM, copies of any modified JLIBSOURCE promptly after modifications are made.

You agree that any information, source code, and object code that you may provide to IBM related to JLIBSOURCE, modified JLIBSOURCE, JLIBRARIES, and modified JLIBRARIES is non-confidential and you grant to IBM, its affiliates, and its licensees a non-exclusive, fully-paid up, irrevocable license to use, make, have made, perform, prepare and have prepared derivative works, and sublicense such information, source code, and object code. You warrant that such information, source code, and object code is original to you and that you have the right to grant the licenses set out in this paragraph

LIMITATIONS: You acknowledge that under this Agreement you obtain only the rights to JLIBSOURCE, modified JLIBSOURCE, and modified JLIBRARIES as expressly stated in this Agreement. No right, title or interest in or to any copyrights, trademarks patents or other proprietary or intellectual rights relating to JLIBSOURCE, modified JLIBSOURCE, and modified JLIBRARIES is transferred to you.

TERM AND TERMINATION: You may terminate your license to JLIBSOURCE, modified JLIBSOURCE, and modified JLIBRARIES at any time by written notice to IBM and by destroying all your copies of JLIBSOURCE, modified JLIBSOURCE, and modified JLIBRARIES. IBM may immediately terminate your license at any time if you fail to comply with the terms of this Agreement or if so required by a court of competent jurisdiction.

DISCLAIMER OF WARRANTY AND LIABILITY:

JLIBSOURCE IS PROVIDED "AS IS" WITHOUT WARRANTY OR CONDITION OF ANY KIND EITHER EXPRESSED, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. THE ENTIRE RISK OF THE USE OR PERFORMANCE OFJLIBSOURCE REMAINS WITH YOU. IN NO EVENT WILL IBM BE LIABLE FOR ANY LOST PROFITS, LOST SAVINGS, LOST REVENUE, LOST DATA OR FOR ANY DIRECT, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF IBM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES OR DAMAGES. IN ADDITION, IBM WILL NOT BE LIABLE FOR ANY DAMAGES CLAIMED BY YOU BASED ON ANY THIRD PARTY CLAIM.

Some jurisdictions do not allow the exclusion of implied warranties, or the limitation of consequential damages, so the above may not apply to you.

Note to U.S. Government Users - Documentation related to Restricted Rights - Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

Although IBM may try to answer technical support questions you have regarding your use of JLIBSOURCE, any such assistance does not obligate IBM to provide support or maintenance services for JLIBSOURCE.

GENERAL:

All terms and conditions, which by their nature should survive the termination or expiration of this Agreement, shall survive. If any provision of this Agreement is declared unenforceable at law, in whole or in part, the rest of the provisions of the Agreement shall remain in effect.

Only a written Agreement signed by both of us can modify this Agreement.

This Agreement is entered into in the State of New York, USA, and governed by the laws of the State of New York without regard to its conflict of law principles.

Index