

Jikes RVM User Guide

Jikes RVM Contributors and Core Team

2016-02-18

Contents

I	Care and Feeding	7
1	Quick Start Guide	8
2	Get the Source	9
2.1	Download a Release	9
2.2	Use Git	9
3	Building Jikes RVM	11
3.1	Overview	11
3.1.1	Compiling the source code	11
3.1.2	Generating source code	11
3.1.3	Bootstrapping Jikes RVM	12
3.1.4	Class libraries	12
3.2	Target Requirements	12
3.2.1	Architectures	12
3.2.2	Operating Systems	13
3.2.3	Support Matrix	13
3.3	Tool Requirements	14
3.4	Instructions	14
3.4.1	Defining Ant properties	14
3.4.2	Selecting a Configuration	16
3.4.3	Fetching Dependencies	16
3.4.4	Building Jikes RVM	17
3.4.5	Running Jikes RVM	17
3.5	Building Patched Versions	17
3.6	Cross-Platform Building	17
3.7	Primordial Class List	18
3.8	Using buildit	19
3.8.1	Examples	19
4	Configuring Jikes RVM	22
4.1	Jikes RVM Configurations	22
4.1.1	Logical Configurations	22
4.1.2	Configurations in Depth	24

5	Debugging Jikes RVM	27
5.1	General debugging tips	27
5.1.1	Assertions	27
5.1.2	Options	27
5.1.3	Debugger Thread	27
5.1.4	Shutdown hooks	28
5.1.5	Tests	28
5.2	Tools	28
5.2.1	GDB	28
5.2.2	rdb	28
5.2.3	Other Tools	28
5.3	Debugging Optimizing Compiler Problems	29
5.3.1	Deadlocks	29
5.3.2	Excluding Garbage Collection problems	29
5.4	GDB Stack Walking	29
6	Experimental Guidelines	32
6.1	Which boot image should I use?	32
6.2	Compiler Replay	32
6.2.1	Generate Advice	33
6.2.2	Executing with advice	33
6.2.3	Verbosity	34
6.3	Measuring GC performance	34
6.4	Jikes RVM is really slow! What am I doing wrong?	35
6.5	Stability of Jikes RVM	36
7	Modifying Jikes RVM	37
7.1	Adding command line options	37
7.1.1	MMTk options	37
7.1.2	VM subsystem options	38
7.1.3	Core VM options	39
7.2	Adding a new garbage collector	41
7.2.1	Overview	41
7.2.2	Prerequisites	41
7.2.3	Cloning the MarkSweep GC	42
7.2.4	Making it Prettier	43
7.2.5	Beyond BaseBaseMyGC	43
7.2.6	What Next?	44
7.3	Coding Conventions	44
7.3.1	Assertions in Jikes RVM and MMTk	44
7.3.2	Assertions in the MMTk Test Harness	44
7.3.3	Error Handling	45
7.4	Coding Style	45
7.4.1	File Headers	45
7.4.2	Coding style description	46
7.4.3	Javadoc requirements	47
7.5	Compiler DNA	47
7.6	Editing Jikes RVM in an IDE	48
7.6.1	Editing JikesRVM in Eclipse	48
7.6.2	Setup for easier compliance with the Checkstyle rules	49

7.6.3	Editing JikesRVM in NetBeans	51
8	Profiling Applications with Jikes RVM	53
8.1	Instrumented Event Counters	54
8.1.1	Existing Instrumentation Phases	54
8.1.2	Writing new instrumentation phases	55
9	Running Jikes RVM	57
9.1	Standard Command-Line Options	57
9.2	Non-Standard Command-Line Options	57
9.2.1	Core Non-Standard Command-Line Options	58
9.2.2	Memory Non-Standard Command-Line Options	58
9.2.3	Subsystem Non-Standard Command-Line Options	58
9.3	Running Jikes RVM with Valgrind	59
10	Testing Jikes RVM	61
10.1	Unit Tests	61
10.2	Functional and Performance Tests	61
10.2.1	Ant properties	62
10.2.2	Defining a test-run	62
10.2.3	Excluding tests	64
10.2.4	Executing a test-run	64
10.2.5	Jenkins integration	64
10.3	External Test Resources	65
10.3.1	Open Source Benchmarks	65
10.3.2	Proprietary Benchmarks	67
10.4	Test Run Descriptions	68
10.4.1	pre-commit	68
10.4.2	core	68
10.4.3	sanity	69
10.4.4	Other test runs	69
10.4.5	Summary	69
11	The MMTk Test Harness	72
11.1	Overview	72
11.2	Standalone	72
11.3	In Eclipse	72
11.4	Test harness options	75
11.5	Scripts	77
11.6	Scripting language	77
11.6.1	Basics	77
11.6.2	Syntax	79
11.7	MMTk Unit Tests	80
II	Architecture	81
12	Adaptive Optimization System	83
12.1	AOS Controller	85
12.2	Cost Benefit Model	86

12.3	Jikes RVM's compilers	87
12.4	Life Cycle of a Compiled Method	89
12.5	Logging and Debugging	90
12.6	Threading and Yieldpoints	93
13	Compilers	94
13.1	Baseline Compiler	94
13.1.1	General Architecture	94
13.1.2	GC Maps	95
13.2	Optimizing Compiler	95
13.2.1	Method Compilation	95
13.2.2	IR	97
13.2.3	BURS	100
13.2.4	OptTestHarness	102
14	Core Runtime Services	104
14.1	Object Model	105
14.1.1	Overview	105
14.1.2	Object Header	105
14.1.3	Field Layout	106
14.2	Class and Code Management	107
14.2.1	Class Loading	107
14.2.2	Code Management	108
14.3	Thread Management	110
14.3.1	Tracking the Thread State	110
14.3.2	Blocking and Handshaking	114
14.3.3	The Monitor API	115
14.3.4	Thin and Biased Locking	116
14.4	JNI	118
14.4.1	Overview	118
14.4.2	JNI Functions	118
14.4.3	Invoking Native Methods	118
14.4.4	Interactions with Threading	119
14.4.5	Missing Features	119
14.4.6	Things JNI Can't Handle	119
14.5	Exception Management	120
14.6	Bootstrap	121
14.6.1	Memory Map	121
14.7	Calling Conventions	123
14.7.1	Architecture-independent concepts	123
14.7.2	Architecture-specific calling conventions	124
14.8	VM Callbacks	124
14.8.1	An Example: Modifying SPECjvm98 to Report the End of a Run	125
14.8.2	Another Example: Directing a Recompilation of All Meth- ods During the Application's Execution	126

15 Magic	127
15.1 Compiler Intrinsic	128
15.2 Unboxed Types	128
15.3 Raw Memory Access	129
15.4 Uninterruptible Code	130
16 MMTk	133
16.1 Anatomy of a Garbage Collector	134
16.1.1 Structure of a Plan	134
16.1.2 Policies	135
16.1.3 Allocation	136
16.1.4 Collection	138
16.2 Memory Allocation in Jikes RVM	141
16.2.1 Memory Manager Interface	141
16.2.2 Compiler integration	143
16.3 Scanning Objects in Jikes RVM	144
16.3.1 JikesRVM to MMTk Interface	144
16.3.2 Hand Inlining	145
16.3.3 Specialized Scanning	145
16.4 Using GCspy	146
16.4.1 The GCspy Heap Visualisation Framework	146
16.4.2 Installation of GCspy with Jikes RVM	147
16.4.3 Command line arguments	147
16.4.4 Writing GCspy drivers	148
III MMTk Tutorial	151
17 Preliminaries	153
17.1 Getting MMTk and Jikes RVM and Eclipse working	153
17.2 Creating The Base Tutorial Collector	153
18 Building a Mark-sweep Collector	155
18.1 Free-list Allocation	155
18.2 Mark-Sweep Collection	156
18.3 Optimized Mark-sweep Collection	160
19 Building a Hybrid Collector	162
19.1 Add a Copying Nursery	162
19.2 Add nursery allocation	163
19.3 Add copying to the collector	164
19.4 Make necessary changes to TutorialTraceLocal	164

The User Guide provides JikesTM RVM information that is not typically covered in published papers. For high-level overviews, algorithms, and structures, you will find the published papers to be the best starting place. The User Guide supplements these Jikes RVM papers, focusing on implementation details of how to build, run, and add functionality to the system.

You may find sections of the User 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 core mailing list or open a pull request on GitHub. Constructive criticism will be cheerfully accepted.

- Care and Feeding: The guide to practical aspects of building, testing, debugging and evaluating Jikes RVM.
- Architecture: The guide to the major architectural decisions of Jikes RVM.
- MMTk Tutorial: A simple tutorial to building a collector with MMTk.

Part I

Care and Feeding

Chapter 1

Quick Start Guide

On Ubuntu amd64,

```
apt-get install git ant gcc g++ gcc-multilib g++-multilib  
bison automake gettext libtool
```

Then check your Java version. If you are using JDK 7, either ensure you are building a Jikes RVM version $\geq 3.1.3$ or switch to JDK 6.

Then,

```
git clone https://github.com/JikesRVM/JikesRVM.git  
cd jikesrvm
```

You can then build with Ant using

```
ant -Dconfig.name=prototype-opt  
./dist/prototype-opt_x86_64-linux/rvm -version
```

or with buildit

```
bin/buildit -j /usr/lib/jvm/default-java localhost  
prototype-opt  
./dist/prototype-opt_x86_64-linux/rvm -version
```

Chapter 2

Get the Source

The source code for the Jikes RVM is stored in a Git repository. You can browse the online Git repository at <https://github.com/JikesRVM/JikesRVM>. A developer can either work with the version control system or download one of the releases. If you are interested in doing development of Jikes RVM you should probably use Git instead of downloading a release.

2.1 Download a Release

Major and minor releases of Jikes RVM occur at regular intervals. These releases are archived in the file download area in either tar-gzip (jikesrvm-`[version]`.tar.gz) or tar-bzip2 (jikesrvm-`[version]`.tar.bz2) format. Use your web browser to download the latest version of Jikes RVM then to extract the tar-gzip archive type:

```
$ tar xvzf jikesrvm-<version>.tar.gz
```

or for the tar-bzip2 archive type:

```
$ tar xvjf jikesrvm-<version>.tar.bz2
```

2.2 Use Git

The source code for Jikes RVM is stored in a Git repository. Git and other distributed revision control systems (e.g. Mercurial) are very different from centralized version control systems like CVS and Subversion. If you are not familiar with Git, you can find instructions on Git use at <http://www.git-scm.com/doc>. There is also a Git book.

After installing Git the current version of source can be downloaded via:

```
$ git clone https://github.com/JikesRVM/JikesRVM.git
```

This will clone the Jikes RVM repository into the newly created directory `jikesrvm`.

If you need a specific version, it is recommended to clone the complete repository nonetheless. You can then switch to a specific release, e.g. 2.4.6, by doing the following:

```
$ cd jikesrvm  
$ git checkout 2.4.6
```

If you are a not core developer you will not be able to push changes to the main Jikes RVM repository directly. If you want to contribute to the Jikes RVM, please take a look at [this page](#).

Chapter 3

Building Jikes RVM

This guide describes how to build Jikes RVM. The first section is an overview of the Jikes RVM build process and this is followed by your system requirements and a detailed description of the steps required to build Jikes RVM.

Once you have things working, as described below, the buildit script will provide a fast and easy way to build the system. We recommend you get things working as described below first, so you can be sure you've met the requisite dependencies etc.

3.1 Overview

To avoid problems with the build, make sure that the path to the Jikes RVM source code doesn't contain any whitespace.

If you run into trouble when building Jikes RVM, don't hesitate to ask for help on the researchers mailing list.

3.1.1 Compiling the source code

The majority of Jikes RVM is written in Java and will be compiled into class files just as with other Java applications. There is also a small portion of Jikes RVM that is written in C that must be compiled with a C compiler such as gcc. Jikes RVM uses Ant version 1.7.0 or later as the build tool that orchestrates the build process and executes the steps required in building Jikes RVM.

Jikes RVM requires a complete install of ant, including the optional tasks. These are present if you download and install ant manually. Some Linux distributions have decided to break ant into multiple packages. So if you are installing on a platform such as Debian you may need to install another package such as 'ant-optional'.

3.1.2 Generating source code

The build process also generates Java and C source code based on build time constants such as the selected instruction architecture, garbage collectors and compilers. The generation of the source code occurs prior to the compilation phase.

3.1.3 Bootstrapping Jikes RVM

Jikes RVM compiles Java class files and produces arrays of code and data. To build itself Jikes RVM will execute on an existing Java Virtual Machine and compiles a copy of its own class files into a boot image for the code and data using the boot image writer tool. The set of files compiled is called the Primordial Class List. The boot image runner is a small C program that loads the boot image and transfers control flow into Jikes RVM.

3.1.4 Class libraries

The Java class library is the mechanism by which Java programs communicate with the outside world. Jikes RVM has configurable class library support, the most mature of which is the GNU Classpath class library.

For GNU Classpath, the developer can either specify a particular version of GNU Classpath to use. By default the build process will download and build GNU Classpath.

Previous releases of the Jikes RVM had support for the Apache Harmony class library. This is no longer developed or supported because Apache Harmony development was stopped. Support for OpenJDK is planned, but not yet implemented.

3.2 Target Requirements

3.2.1 Architectures

The PowerPC (or ppc) and ia32 instruction set architectures are supported by Jikes RVM.

Intel's Instruction Set Architectures (ISAs) get known by different names:

- IA-32 is the name used to describe processors such as 386, 486 and the Pentium processors. It is popularly called x86 or sometimes in our documentation as x86-32.
- IA-32e is the name used to describe the extension of the IA-32 architecture to support 8 more registers and a 64-bit address space. It is popularly called x86_64 or AMD64, as AMD chips were the first to support it. It is found in processors such as AMD's Opteron and Athlon 64, as well as in Intel's own Pentium 4 processors that have EM64T in their name.
- IA-64 is the name of Intel's Itanium processor ISA.

Jikes RVM currently supports the IA-32 ISA and work on IA32-e is in progress. As IA-32e is backward compatible with IA-32, Jikes RVM can be built and run upon IA-32e processors. The IA-64 architecture supports IA-32 code through a compatibility mode or through emulation and Jikes RVM should run in this configuration. Native IA-64 is not supported.

On PowerPC, only big endian is supported.

target.name	OS	ISA	Address size	Status
ia32-linux	Linux	IA32	32 bits	OK
ia32-osx	OS X	IA32	32 bits	???
ia32-solaris	Solaris	IA32	32 bits	???
ia32-cygwin	Windows	IA32	32 bits	NYI
x86_64-linux	Linux	IA32	32 bits	OK
x86_64-osx	OS X	IA32	32 bits	???
x86_64_m64-linux	Linux	IA32e	64 bits	WIP
x86_64_m64-osx	OS X	IA32e	64 bits	???
ppc32-linux	Linux	ppc32 (big e.)	32 bits	???
ppc64-linux	Linux	ppc64 (big e.)	64 bits	OK

Table 3.1: platform support matrix

3.2.2 Operating Systems

Jikes RVM is capable of running on any operating system that is supported by the GNU Classpath library, low level library support is implemented and memory layout is defined. The low level library support includes interaction with the threading and signal libraries, memory management facilities and dynamic library loading services. The memory layout must also be known, as Jikes RVM will attempt to locate the boot image code and data at specific memory locations. These memory locations must not conflict with where the native compiler places it's code and data. Operating systems that are known to work include Linux and OS X. At one stage a port to win32 was completed but it was never integrated into the main Jikes RVM codebase. AIX was supported previously but support has been removed due to lack of demand. The same applies for support of Mac OS on PPC.

Note: Current implementation of Jikes RVM implies that system native libraries (like GTK+) have been compiled with frame pointers. Most of Linux distribution have frame pointers enabled in most of the packages, but some explicitly use `-fomit-frame-pointer` thus producing the library that can't be used with Jikes RVM.

3.2.3 Support Matrix

The platform support matrix table details the targets that have historically been supported and the current status of the support. The target.name column is the identifier that Jikes RVM uses to identify this target. ??? means that we don't have regression machines for this platform so the Jikes RVM team can't guarantee that the target works at a given point in time. We rely on the community to provide a Jikes RVM implementation on these platforms.

x86_64 is currently only supported using the legacy 32bit addressing mode and instructions. You need to install the 32-bit versions of the required libraries to build and use the x86_64 configurations.

Note that building on Windows is currently not supported. All previous attempts at building on Windows natively (i.e. without cygwin) used the Apache Harmony classlibrary whose development has been discontinued. Support for building with cygwin is not yet implemented.

3.3 Tool Requirements

Java Virtual Machine Jikes RVM requires an existing Java Virtual Machine that conforms to Java 6.0 such as Oracle JDK 1.6, OpenJDK/IcedTea 6 or IBM SDK 6.0. We also aim to support the Java 7.0-conformant and Java 8.8-conformant versions of these virtual machines.

Some Java Virtual Machines are unable to cope with compiling the Java class library so it is recommended that you install one of the above mentioned JVMs if they are not already installed on your system. The remaining build instructions assume that a suitable Java Virtual Machine is on your path. You can run `java -version` to check you are using the correct JVM.

Ant Ant version 1.7.0 or later is the tool required to orchestrate the build process. You can download and install the Ant tool from its Apache homepage if it is not already installed on your system. The remaining build instructions assume that `$ANT_HOME/bin` is on your path and points to a full Ant installation (i.e. including the optional tasks). You can run `ant -version` to check you are running the correct version of ant.

C compilers The Jikes RVM build assumes that the GNU Compiler Collection is present on the system. Most modern *nix environments satisfy this requirement. Clang should also work but is untested.

Bison As part of the build process, Jikes RVM uses the bison tool which should be present on most modern *nix environments.

Perl Perl is trivially used as part of the build process but this requirement may be removed in future releases of Jikes RVM. Perl is also used as part of the regression and performance testing framework.

Awk GNU Awk is required as part of the regression and performance testing framework but is not required when building Jikes RVM.

Extra tools recommended for Solaris `pkg-get` will greatly simplify installing GNU packages on Solaris. Our patches require that GNU patch is picked up in preference to Sun's. You can create a symbolic link to `/usr/bin/gpatch` from `/opt/csw/bin/patch` and make sure `/opt/csw/bin` is in your path before `/usr/bin` in order to achieve this.

3.4 Instructions

3.4.1 Defining Ant properties

There are a number of ant properties that are used to control the build process of Jikes RVM. These properties may either be specified on the command line by `-Dproperty=variable` or they may be specified in a file named `.ant.properties` in the base directory of the jikesrvm source tree. The `.ant.properties` file is a standard Java property file with each line containing a `property=variable` and comments starting with a `#` and finishing at the end of the line.

Property	Description	Default
host.name	The name of the host environment used for building Jikes RVM. The host environment defines the paths to the tools used during the build, e.g. the path to the C compiler. The name should match one of the files located in the build/hosts/ directory minus the <code>.properties</code> extension.	None
target.name	The name of the target environment for Jikes RVM. The name should match one of the files located in the build/targets/ directory minus the <code>.properties</code> extension. This should only be specified when cross compiling the Jikes RVM. See Cross-Platform Building for a detailed description of cross compilation.	<code>\${host.name}</code>
config.name	The name of the configuration used when building Jikes RVM. The name should match one of the files located in the build/configs/ directory minus the <code>.properties</code> extension. This setting is further described in the section Configuring Jikes RVM.	None
patch.name	An identifier for the current patch applied to the source tree. See Building Patched Versions for a description of how this fits into the standard usage patterns of Jikes RVM.	“ ”
components.dir	The directory where Ant looks for external components when building Jikes RVM.	<code>\${jikesrvm.dir}/components</code>
dist.dir	The directory where Ant stores the final Jikes RVM runtime.	<code>\${jikesrvm.dir}/dist</code>
build.dir	The directory where Ant stores the intermediate artifacts generated when building the Jikes RVM.	<code>\${jikesrvm.dir}/target</code>
components.-cache.dir	The directory where Ant caches downloaded components. If you explicitly download a component, place it in this directory.	(Undefined, forcing download)
require.rvm-unit-tests	If set to <code>true</code> , run unit tests on the built Jikes RVM image. Use with care as it will significantly increase build times for configurations that are compiled using a non-optimizing compiler (see below).	(Undefined, tests are not run)
require.checkstyle	Only useful if you want to contribute changes to the Jikes RVM. If set to <code>true</code> , run checkstyle during the build to check for violations of the Jikes RVM Coding Style and Coding Conventions for assertions.	(Undefined, no checks run)
rvm.debug-symbols	If set to <code>true</code> , build the Jikes RVM with debug symbols for the bootloader code and the code in the bootimage. Note: this is not enabled by default because it causes build failures for configurations that build the bootimage with the optimizing compiler (see RVM-1084).	(Undefined, no symbols built)
protect.config-files	Define this property if you do not want the build process to update configuration files when auto downloading components.	(Undefined)

Table 3.2: Commonly used Ant properties for Jikes RVM

At a minimum it is recommended that the user specify the `host.name` property in the `.ant.properties` file.

The configuration files in `build/targets/` and `build/hosts/` are designed to work with a typical install but it may be necessary to override specific properties. The easiest way to achieve this is to specify the properties to override in the `.ant.properties` file.

3.4.2 Selecting a Configuration

A configuration in terms of Jikes RVM is the combination of build time parameters and component selection used for a particular Jikes RVM image. The section Configuring Jikes RVM section describes the details of how to define a configuration. Typical configuration names include:

- **production:** This configuration defines a fully optimized version of the Jikes RVM.
- **development:** This configuration is the same as production but with debug options enabled. The debug options perform internal verification of Jikes RVM which means that it builds and executes more slowly.
- **prototype:** This configuration is compiled using an unoptimized compiler and includes minimal components which means it has the fastest build time.
- **prototype-opt:** This configuration is compiled using an unoptimized compiler but it includes the adaptive system and optimizing compiler. This configuration has a reasonably fast build time.

If a user is working on a particular configuration most of the time they may specify the `config.name` ant property in `.ant.properties` otherwise it should be passed in on the command line `-Dconfig.name=...`

3.4.3 Fetching Dependencies

The Jikes RVM has a build time dependency on the GNU Classpath class library and depending on the configuration may have a dependency on GCSPy. The build system will attempt to download and build these dependencies if they are not present or are the wrong version.

To just download and install the GNU Classpath class library you can run the command `"ant -f build/components/classpath.xml"`. After this command has completed running it should have downloaded and built the GNU Classpath class library for the current host. See the Using GCSPy page for directions on building configurations with GCSPy support.

If you wish to manually download components (for example you need to define a proxy, so ant is not correctly downloading), you can do so and identify the directory containing the downloads using `-Dcomponents.cache.dir=<download directory>` when you build with ant.

3.4.4 Building Jikes RVM

The next step in building Jikes RVM is to run the ant command `ant` or `ant -Dconfig.name=...`. This should build a complete RVM runtime in the directory `${dist.dir}/${config.name}_${target.name}`. A complete list of documented targets can be listed by executing the command `ant -projecthelp`.

3.4.5 Running Jikes RVM

Jikes RVM can be executed in a similar way to most Java Virtual Machines. The difference is that the command is `rvm` and resides in the runtime directory (i.e. `${dist.dir}/${config.name}_${target.name}`). See Running Jikes RVM for a list of command line options.

3.5 Building Patched Versions

As part of the research process there will be a need to evaluate a set of changes to the source tree. To make this process easier the property named `patch.name` can be set to a non-empty string. This will cause the output directory to have the name `${config.name}_${target.name}_${config.variant}` rather than `${config.name}_${target.name}`, thus making it easy to differentiate between the patched and unpatched runtimes.

The following steps will create a runtime without the patch in `dist/prototype-ia32-linux` and a runtime with the patch applied in `dist/prototype-ia32-linux.ReadBarriers`:

```
% cd $RVMROOT
% ant -Dconfig.name=prototype -Dhost.name=ia32 -linux
% patch -p0 < ReadBarriers.diff
% ant -Dconfig.variant=ReadBarriers -Dconfig.name=
    prototype -Dhost.name=ia32 -linux
% patch -R -p0 < ReadBarriers.diff
```

The `config.variant` property is also supported and reported as part of the test infrastructure.

3.6 Cross-Platform Building

The JikesTM RVM build process consists of two major phases: the building of a *boot image*, and the building of a *bootloader*. The boot image is built using a JavaTM 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 building the boot image can be time-consuming, you may prefer to build the boot image on a faster machine than the target platform. You may also be porting Jikes RVM to a target platform that lacks tools such or whose development environment is otherwise unpleasant. To cross-build, simply set your `host.name` and `target.name` properties to different values.

For example, to build the prototype configuration for AIXTM on a Linux host:

```
% cd $RVMROOT
% ant -Dconfig.name=prototype -Dhost.name=ia32-linux -
      Dtarget.name=ppc32-aix cross-compile-host
```

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

```
% cd $RVMROOT
% ant -Dconfig.name=prototype -Dhost.name=ppc32-aix cross
      -compile-target
```

After the script has completed successfully, you should be able to run Jikes 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.

Dependencies

To compile the boot image on the host system you will also need to have built any dependencies on the target machine and then copied them to the host machine. You will also need to add an appropriate line into your `${components.dir}components.properties` file such as the following (if the target system was ppc32-linux):

```
ppc32-linux.classpath.lib.dir=path/to/components/
classpath/95/ppc32-linux/lib
```

It may be possible to simply build the dependencies on the host machine. Modify the `${components.dir}/components.properties` so that the dependency property for target machine maps to the same value as the dependency property on the host machine. This works at the current time but may fail in the future if classpath changes the API between platforms. i.e.

```
ppc32-linux.classpath.lib.dir=path/to/components/
classpath/95/ia32-linux/lib
```

3.7 Primordial Class List

The primordial class list indicates which classes should be compiled and baked into the boot image. The bare minimum set of classes needed in the primordial list includes:

- All classes that are needed to load a class from the file system. The class may need to be loaded as a single class file or out of a jar. Failing this there will be an infinite regress on the first class load.
- All classes that are needed by the baseline compiler to compile any method. Failing this we regress when attempting to compile a method so we can execute it.

- Enough of the core VM services and data structures, and class library (java.*) to support the above. This includes threading, memory management, runtime support for compiled code, etc.

For increased performance and decreased startup time it is possible to include extra classes that are expected to be needed, i.e. the optimizing compiler or the adaptive system. There are some pieces of these components that would be awkward to load dynamically (there's a core subset of the opt compiler, the classes in the `org.jikesrvm.compilers.opt.runtimesupport` packages, that must be loaded and fully compiled before any opt-compiled code can be allowed to executed), but it's theoretically possible to do so.

If you took a full closure of the classes referenced by things that have to be in the bootimage you'd actually end up with a lot more in the bootimage than we currently have. The culprit here would I think mainly be java.* classes that we need in the bootimage, but only use in restricted ways, so we don't actually have to drag in everything they depend on to meet the "real" constraints of what has to go in the bootimage. It is unknown how much difference there is between hand-crafted include lists and what an automated tool would discover.

3.8 Using buildit

The buildit script is a handy way to build and test the system. It has countless features and options to make building and testing really easy, particularly in a multi-machine environment, where you edit on one machine and build and test on others. If you really want to get the most of it, take a look at all the options by running:

```
bin/buildit -h
```

...or read the script itself.

3.8.1 Examples

Here we just provide a hand full of examples of how it is often used, first for building and secondly for testing (which includes building). Please add to the list if you have other really useful ways of using it. In the examples below, we'll use three hypothetical hosts: **habanero** (your desktop), **jalapeno** (a remote x86 machine) and **chipotle** (a remote PowerPC machine).

Simple Builds

To build a production image on your desktop, habanero, do the following:

```
bin/buildit habanero production
```

Or equivalently:

```
bin/buildit localhost production
```

To build a production image on the remote machine jalapeno, do the following:

```
bin/buildit jalapeno production
```

Cross Platform Building

To build a production image on the remote PowerPC machine chipotle, do the following:

```
bin/buildit chipotle production
```

Since building on a PowerPC machine can take a long time, you might prefer to build on your x86 machine jalapeno and cross-build to chipotle. In that case you would just do the following:

```
bin/buildit jalapeno -c chipotle production
```

In each case, buildit figures out the host types by interrogating them and does the right thing (forcing a PPC build on the x86 host jalapeno since you've told it you want a build for chipotle, which it knows is PPC). Buildit caches the host information, and will prompt you the first time it encounters a new host.

Full Build Specification

If you want to specify the build fully, you can do something like this:

```
bin/buildit jalapeno FastAdaptive MarkSweep
```

If you want to specify multiple different GCs you could do:

```
bin/buildit jalapeno FastAdaptive MarkSweep SemiSpace  
GenMS
```

which would build all three configurations on jalapeno.

Profiled Builds

Jikes RVM has the capacity to profile the boot image and then re-build an optimized boot image based on the profiles. This process takes a little longer, but results in measurably faster builds, and so should be used when doing performance testing. Buildit lets you do this trivially:

```
bin/buildit jalapeno --profile production
```

Testing

Jikes RVM currently has a notion of a **"test-run"**, which defines a complete test scenario, including tests and builds. An example is *pre-commit*, which runs a small suite of pre-commit tests. It also has the notion of a **"test"**, which just specifies a particular set of tests, not the full scenario. An example is *dacapo*, which just runs the DaCapo test suite (see the testing/tests directory for the available tests).

Running a test run

To run the pre-commit test-run on your host jalapeno just do:

```
bin/buildit jalapeno --test-run pre-commit jalapeno
```

Running a test

To run the dacapo tests against a production on the host jalapeno, do:

```
bin/buildit jalapeno -t dacapo production
```

To run the dacapo tests against a FastAdaptive MarkSweep build, on the host jalapeno, do:

```
bin/buildit jalapeno -t dacapo FastAdaptive MarkSweep
```

To run the dacapo and SPECjvm98 tests against production on the host jalapeno, do:

```
bin/buildit jalapeno -t dacapo -t SPECjvm98 production
```

Chapter 4

Configuring Jikes RVM

The build process requires a number of build time parameters that specify the features and components for a Jikes RVM build. Typically the build parameters are defined within a property file located in the `build/configs` directory. The following table defines the parameters for the build configuration.

4.1 Jikes RVM Configurations

A typical user will use one of the existing build configurations and thus the build system only requires that the user specify the `config.name` property. The name should match one of the files located in the `build/configs/` directory minus the `'.properties'` extension.

4.1.1 Logical Configurations

There are many possible Jikes RVM configurations. Therefore, we define four "logical" configurations that are most suitable for casual or novice users of the system. The four configurations are:

- **prototype:** A simple, fast to build, but low performance configuration of Jikes RVM. This configuration does not include the optimizing compiler or adaptive system. Most useful for rapid prototyping of the core virtual machine.
- **prototype-opt:** A simple, fast to build, but low performance configuration of Jikes RVM. Unlike prototype, this configuration does include the optimizing compiler and adaptive system. Most useful for rapid prototyping of the core virtual machine, adaptive system, and optimizing compiler.
- **development:** A fully functional configuration of Jikes RVM with reasonable performance that includes the adaptive system and optimizing compiler. This configuration takes longer to build than the two prototype configurations.
- **production:** The same as the development configuration, except all assertions are disabled. This is the highest performance configuration of

Property	Description	Default
config.name	A unique name that identifies the set of build parameters.	None
config.bootimage.compiler	Parameter selects the compiler used when creating the bootimage. Must be either opt or base.	base
config.bootimage.compiler.args	Parameter specifies any extra args that are passed to the bootimage compiler.	""
config.bootimage.writer.args	Parameter specifies any extra args that are passed to the bootimage writer.	""
config.runtime.compiler	Parameter selects the compiler used at runtime. Must be either opt or base.	base
config.include.aos	Include the adaptive system if set to true. Parameter will be ignored if config.runtime.compiler is not opt.	false
config.mmtk	The MMTk properties file to use. MMTk property files are located in the build/mmtk directory.	default
config.mmtk.plan	The name of the GC plan to use for the build. See MMTk for more details.	None
config.default-heapsize.initial	Parameter specifying the default initial heap size in MB.	20
config.default-heapsize.maximum	Parameter specifying the default maximum heap size in MB.	100
config.assertions	Parameter specifies the level of assertions in the code base. Must be one of extreme, normal or none.	normal
config.stress-gc-interval	The build will stress test the gc subsystem if set to a positive value. The value indicates the number of allocations between collections	0
config.include.perfevent	Set to true to build Jikes RVM with support for performance counters.	false
config.include.gcspy	Set to true to build Jikes RVM with GCspy support. See Using GCspy for more details.	false
config.include.gcspy-client	Set to true to bundle the GCspy client with the Jikes RVM build. Parameter will be ignored if config.include.gcspy is not true.	false
config.include.gcspy-stub	Set to true to use the GCspy stub rather than the real GCspy component. Parameter will be ignored if config.include.gcspy is not true.	false
config.include.all-classes	Include all the Jikes RVM classes in the bootimage if set to true.	false

Table 4.1: Parameters for build configurations

Jikes RVM and is the one to use for benchmarking and performance analysis. Build times are similar to the development configuration.

The mapping of logical to actual configurations may vary from release to release. In particular, it is expected that the choice of garbage collector for these logical configurations may be different as MMTk evolves.

Logical configurations that are not mentioned here are not recommended for novice users of the system.

4.1.2 Configurations in Depth

Most standard Jikes RVM configuration files follow the following naming scheme: *[ExtremeAssertions]* (**Base** | **Full** | **Fast**) (Base | Adaptive) *<garbage collector>* where

- *ExtremeAssertions* is optional. Its presence indicates that the `config.assertions` configuration parameter is set to `extreme`. This turns on a number of expensive assertions.
- **Base** | **Full** | **Fast** determines the performance of the Jikes RVM boot image. **Base** denotes baseline compiler and enabled assertions, **Full** denotes optimizing compiler and enabled assertions, **Fast** denotes optimizing compiler and disabled assertions. Note that **Fast** is exclusive with *ExtremeAssertions* and that **Full** and **Fast** imply that adaptive system and optimizing compiler are included in the image.
- Base | Adaptive denotes whether or not the adaptive system and optimizing compiler are included in the build.
- the *<garbage collector>* is the garbage collection scheme used.

Each version of Jikes RVM provides several garbage collector choices. For a definitive list of garbage collector choices, please refer to the configurations that are shipped with your version of Jikes RVM. If you need a configuration that is not available by default, you can just define your own based on the existing ones (it's easy!).

Some garbage collector suffixes that may be available are:

- "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

- "RefCount" a reference counting collector with synchronous (non-concurrent) cycle collection

For example, to specify a Jikes RVM configuration:

1. with a baseline-compiled boot image,
2. that will compile classes loaded at runtime using the baseline compiler and
3. that uses a non-generational semi-space copying garbage collector,

use the name "**BaseBaseSemiSpace**".

In configurations that include the adaptive system (denoted by "**Adaptive**" in their name), methods are initially compiled by one compiler (by default the baseline compiler) and then online profiling is used to automatically select hot methods for recompilation by the optimizing compiler at an appropriate optimization level.

For example, to build for an adaptive configuration with assertions, where the optimizing compiler is used to compile the boot image and the semi-space garbage collector is used, use the following command:

```
% ant -Dconfig.name=FullAdaptiveSemiSpace
```

Configuration	Description	Potential uses
BaseBaseSomeGC	baseline compiled bootimage with assertions, baseline compiler at runtime	prototyping; debugging of garbage collector SomeGC without having to worry about complexities introduced by compiler optimizations; checking for problems related to uninterruptible code
BaseAdaptiveSomeGC	baseline compiled bootimage with assertions, baseline compiler, adaptive system and optimizing compiler at runtime	prototyping that includes optimizing compiler and adaptive system; debugging of optimizing compiler problems with compiler advice; sanity checks with comparatively short benchmarks; checking for problems related to uninterruptible code
FullAdaptiveSomeGC	bootimage compiled with optimizing compiler and assertions; everything available at runtime	extensive testing including long-running benchmarks; checking for incorrect usage of unboxed types
ExtremeAssertions*	enables all generally useful assertions, including very expensive ones	debugging and testing in special cases
FastAdaptiveSomeGC	bootimage compiled with optimizing compiler; assertions disabled; everything available at runtime	benchmarking
FullBase*	INVALID - Full implies Adaptive	
FastBase*	INVALID - Fast implies Adaptive	
ExtremeAssertionsFast*	INVALID - ExtremeAssertions is incompatible with Fast	

Table 4.2: Example configurations and their uses

LogicalConfiguration	Actual configuration
prototype	BaseBaseGenImmux
prototype-opt	BaseAdaptiveGenImmux
development	FullAdaptiveGenImmux
production	FastAdaptiveGenImmux

Table 4.3: Mapping of logical configurations to actual configurations in Jikes RVM 3.1.3

Chapter 5

Debugging Jikes RVM

This page contains some debugging hints for Jikes RVM. It is assumed that you are familiar with debugging techniques. If you aren't, it is advisable to read a book about the subject.

5.1 General debugging tips

5.1.1 Assertions

All debugging should be done with assertion-enabled builds if possible. You can also try using `ExtremeAssertion` builds.

5.1.2 Options

The Jikes RVM and MMTk provide several options to print out debugging information.

If you're debugging a problem in the optimizing compiler, you can also print out the IR.

You can also use the options to change the behaviour in various ways (e.g. switch off certain optimizations) if you have a suspicion about the causes of the problem.

5.1.3 Debugger Thread

Jikes has an interactive debugger that you can invoke by sending `SIGQUIT` to Jikes while it's running:

```
pkill -SIGQUIT JikesRVM
```

In previous versions of Jikes, that stopped all threads and provided an interactive prompt, but currently it just dumps the state of the VM and continues immediately (that's a known issue: RVM-570). Debug fields in classes
Several classes in the code base provide static boolean fields like `DEBUG` or `VERBOSE` which can be set to get more debugging information.

5.1.4 Shutdown hooks

You can write custom shutdown hooks to dump gathered information when the VM terminates. Note that shutdown hooks won't be run if the VM is terminated via a signal (see RVM-555)

Do not use the `ExitMonitor` from the `Callbacks` class because it's less reliable.

5.1.5 Tests

The test coverage is poor at the moment. Nevertheless, if you're very lucky, one of the smaller test cases will fail. See *Testing Jikes RVM* for details on how to run the tests and define your own.

5.2 Tools

There are different tools for debugging Jikes RVM:

5.2.1 GDB

There is a limited amount of C code used to start Jikes RVM. The `rvm` script will start Jikes RVM using GDB (the GNU debugger) if the first argument is `-gdb`. Break points can be set in the C code, variables, registers can be expected in the C code.

```
rvm -gdb <RVM args> <name of Java application> <
application args>
```

The dynamically created Java code doesn't provide GDB with the necessary symbol information for debugging. As some of the Java code is created in the boot image, it is possible to find the location of some Java methods and to break upon them. To build with debug symbols, you'll need to set the appropriate property as described in *Building Jikes RVM*.

Details of how to manually walk the stack in GDB can be found [here](#).

5.2.2 rdb

`rdb` is a debugger that was developed specifically for Jikes RVM. It allows you to inspect the bootimage. If you're running Mac OS, you can also use it to debug a running Jikes RVM.

5.2.3 Other Tools

Other tools, such as `valgrind`, are occasionally useful in debugging or understanding the behaviour of JikesRVM. The `rvm` script facilitates using these tools with the `'-wrap'` argument.

```
rvm -wrap "<wrapper-script-and-args>" <rest of command
line>
```

For example, `cachegrind` can be invoked by

```
rvm -wrap "/path/to/valgrind --tool=cachegrind" <java -
command-line>
```

The command and arguments immediately after the `-wrap` argument will be inserted into the script on the command line that invokes the boot image runner. One useful variant is

```
rvm -wrap echo <rest of command line>
```

5.3 Debugging Optimizing Compiler Problems

To debug problems in the optimizing compiler, use a configuration whose bootimage is compiled with the baseline compiler and which contains the AOS (prototype-opt, BaseAdaptive*). Faster configurations (such as development) have the drawback of a longer bootimage compilation time which won't be amortized unless the problem occurs late.

It is advisable to use `-X:vm:errorsFatal=true` when debugging optimizing compiler problems. This will prevent the optimizing compiler from reverting to the baseline compiler for certain kinds of errors.

It is strongly recommended to run with advice file generation (see Experimental Guidelines). The produced advice files can then be used to try to reproduce the bug. If this step is successful, the advice files should be minimized to determine the set of methods that cause the failures. This can be done automatically (e.g. via delta debugging) or by hand.

You can also switch on paranoid IR verification in `IR.java`. Note that this is not well tested at the moment because we don't run any regression tests with it. Use a BaseAdaptive* configuration if you switch this on (bootimage builds with the optimizing compiler and paranoid IR fail at the time of this writing).

5.3.1 Deadlocks

To debug a deadlock, run the VM under a time limit and send `SIGQUIT` (to force a thread dump) a few seconds before killing the VM. On Linux, you can use the `timelimit` program (should be available in the repositories for Debian-based distributions).

5.3.2 Excluding Garbage Collection problems

The garbage collectors that are included with the Jikes RVM are generally stable. Therefore, if you see a problem that does not occur during the collection itself, it is likely not a garbage collection problem. You can exclude problems related to garbage collection by building with other collectors. For example, you can choose a collector that doesn't move objects (e.g. MarkSweep) or a collector that doesn't require write barriers (e.g. Immix instead of GenImmix).

5.4 GDB Stack Walking

Sometimes it is desirable to examine the state of the Java stack whilst using GDB to step instructions, break on addresses or watch particular variables.

These instructions are based on an email sent by Martin Hirzel to the rvm-devel list around 15th September 2003. The instructions have been updated by Laurence Hellyer to deal with native threading and renamed RVM classes.

1) To learn about the stack frame layout on IA32, look at `rvm/src/org/jikes-rvm/ia32/StackframeLayoutConstants.java`.
Currently (2009/10/23) the layout is:

```
+4: return address
fp -> 0: caller 's fp
-4: method id
(remember stack grows from high to low)
```

2) To learn how to get the current frame pointer and other context information, look at the `genPrologue()` method in `rvm/src/org/jikesrvm/compiler-s/baseline/ia32/BaselineCompilerImpl.java`. It first retrieves the thread register (`esi` on IA32), which points to an instance of `RVMThread`, and then retrieve fields from that instance.

3) To find the offset of field `RVMThread.framePointer`, add the following lines to the end of `BootImageWriter.main(String[])`:

```
// added to get framePointer offset from RVMThread to
// manually walk stacks in GDB
say("offset of _RVMThread.framePointer==_" +
    ArchEntrypoints.framePointerField.getOffset());
```

Do a build to print this info. On my config I got +148, but this can change between versions

4) To get started, let's examine an example stack that contains methods whose code is in the boot image. We pick one that is likely to be invoked even in a simple hello-world program. In my `RVM.map`, `0x351eae9c` is the address of `org.jikesrvm.mm.mmtk.ReferenceProcessor.growReferenceTable()`;

5) Setting a break point on this address

```
(gdb) break *0x351eae9c
Breakpoint 2 at 0x351eae9c
```

And run the program to the break point

```
Breakpoint 2, 0x351eae9c in ?? ()
```

Step some instructions into the method and then dump the registers

```
(gdb) stepi 30
0x351eaf03 in ?? ()
(gdb) info registers
eax                0x200      512
ecx                0x0        0
edx                0x0        0
ebx                0x7431     29745
esp                0x420e1934   0x420e1934
ebp                0xb0206ed0   0xb0206ed0
esi                0x4100758c   1090549132
edi                0x19c54     105556
```

eip	0x351eaf03	0x351eaf03
eflags	0x202	514
cs	0x17	23
ss	0x1f	31
ds	0x1f	31
es	0x1f	31
fs	0x1f	31
gs	0x37	55

The current FP is stored in RVMThread.framePointer which we found out in 3) is at offset +148. ESI points to the current RVMThread object so we can access the FP value like so:

```
(gdb) x ($esi+148)
0x41007620:      0x420e1954
```

Note that the FP is at a higher address than ESP which is what we would expect

The return address is at FP+4 so to get the return address we can do:

```
(gdb) x (*( $esi+148))+4
0x420e1958:      0x351eadde
```

We can look in RVM.map for the largest method address smaller than 0x351eadde which is org.jikesrvm.mm.mmtk.ReferenceProcessor.addCandidate(java.lang.ref.Reference, org.vmmagic.unboxed.ObjectReference). Examining ReferenceProcessor.java we find that this is the only method that calls growReferenceTable so this is correct

Get the return address from the next frame

```
(gdb) x (*( $esi+148))+4
0x420e1980:      0x352ebd1e
```

Which corresponds to org.jikesrvm.mm.mmtk.ReferenceProcessor.addSoftCandidate(java.lang.ref.SoftReference, org.vmmagic.unboxed.ObjectReference) which is a caller of addCandidate.

We can follow the stack back up to the top where we will read a FP of 0 (look in rvm/src/org/jikesrvm/ia32/StackframeLayoutConstants.java for details)

Chapter 6

Experimental Guidelines

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

6.1 Which boot image should I use?

To make a long story short the best performing configuration of Jikes RVM will almost always be **production**. Unless you really know what you are doing, don't use any other configuration to do a performance evaluation of Jikes RVM. Any boot image you use for performance evaluation must have the following characteristics for the results to be meaningful:

- **config.assertions=none**. 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.
- **config.bootimage.compiler=opt**. Unless this is set, the boot image 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.

6.2 Compiler Replay

The compiler-replay methodology is deterministic and eliminates memory allocation and mutator variations due to non-deterministic application of the adaptive compiler. We need this latter methodology because the non-determinism of the adaptive compilation system makes it a difficult platform for detailed performance studies. For example, we cannot determine if a variation is due to the system change being studied or just a different application of the adaptive compiler. The information we record and use are hot methods and blocks information. We also record dynamic call graph with calling frequency on each edge for inlining decisions.

Note that in December 2011, compiler replay was significantly improved. The notes below apply to the post December 2011 version of replay.

Here is how to use it:

6.2.1 Generate Advice

There are three kinds of advice used by the replay system, each is workload-specific (ie you should generate advice files for each benchmark):

- **Compilation advice (.ca file).** This advice records for every compiled method which compiler (base or opt) and if opt, at which optimization level it should be compiled. Replay compilation will not work without a compilation advice file.
- **Edge counts (.ec file).** This advice captures edge counts generated by the execution of baseline-compiled code. Edge counts are used by the compiler to understand which edges in the control flow graph are hot. At the time of writing, edge counts were measured as contributing about 2
- **Dynamic callgraph (.dc file).** This advice captures the dynamic call graph, which allows the compiler to understand the frequency with which particular call chains occur. This is particularly useful in guiding inlining decisions. At the time of writing the call graph contributes about 8

One way to gather advice is to execute the benchmark multiple times under controlled settings, producing profiles at each execution. Then establish the fastest execution among the set of runs, and choose the profiles associated with that execution as the advice files. A common methodology is to invoke each benchmark 20 times (ie take the best invocation from a set of 20 trials), and in each invocation, run 10 iterations of the benchmark (ie the advice will then capture the warmed-up, steady state of the benchmark). For more advanced methodologies, please refer to current research papers on this topic.

When generating the advice, you will need to use the following command line arguments (typically use all six arguments, so that all three advice files are generated at each invocation):

For adaptive compilation profile

```
-X:aos:enable_advice_generation=true -X:aos:cafo=
my_compiler_advice_file.ca
```

For edge count profile

```
-X:base:profile_edge_counters=true -X:base:
profile_edge_counter_file=my_edge_counter_file.ec
```

For dynamic call graph profile

```
-X:aos:dcfo=my_dynamic_call_graph_file.dc -X:aos:
final_report_level=2
```

6.2.2 Executing with advice

The basic model is simple. At a nominated time in the execution of a program, all methods specified in the .ca advice file will be (re)compiled with the compiler and optimization level nominated in the advice file. Broadly, there are

two ways of initiating bulk compilation: a) by calling the method `org.jikesrvm.adaptive.recompilation.BulkCompile.compileAllMethods()` during execution, and b) by using the `-X:aos:enable_precompile=true` flag at the command line to trigger bulk compilation at boot time. A standard methodology is to use a benchmark harness call back mechanism to call `compileAllMethods()` at the end of the first iteration of the benchmark. At the time of writing this gave performance roughly 2

For **'warmup' replay** (where `org.jikesrvm.adaptive.recompilation.BulkCompile.compileAllMethods()` is called at the end of the first iteration):

```
-X:aos:initial_compiler=base -X:aos:enable_bulk_compile=
  true -X:aos:enable_recompilation=false -X:aos:cafi=
  benchmark.ca -X:vm:edgeCounterFile=benchmark.ec -X:aos
: dcfi=benchmark.dc
```

For **precompile replay** (where bulk compilation occurs at boot time):

```
-X:aos:initial_compiler=base -X:aos:enable_precompile=
  true -X:aos:enable_recompilation=false -X:aos:cafi=
  benchmark.ca -X:vm:edgeCounterFile=benchmark.ec -X:aos
: dcfi=benchmark.dc
```

6.2.3 Verbosity

You can alter the verbosity of the replay behavior with the flag `-X:aos:bulk_compilation_verbosity`, which by default (0) is silent, but will produce more information about the recompilation with values of 1 or 2.

6.3 Measuring GC performance

MMTk includes a statistics subsystem and a harness mechanism for measuring its performance. If you are using the DaCapo benchmarks, the MMTk harness can be invoked using the `'-c MMTkCallback'` command line option, but for other benchmarks you will need to invoke the harness by calling the static methods

```
org.mmtk.plan.Plan.harnessBegin()
org.mmtk.plan.Plan.harnessEnd()
```

at the appropriate places. Other command line switches that affect the collection of statistics are

Option	Description
<code>-X:gc:printStats=true</code>	Print statistics for each mutator/gc phase during the run
<code>-X:gc:xmlStats=true</code>	Print statistics in an XML format (as opposed to human-readable format)
<code>-X:gc:verbose</code>	This is incompatible with MMTk's statistics system.
<code>-X:gc:variableSizeHeap=false</code>	Disable dynamic resizing of the heap

Unless you are specifically researching flexible heap sizes, it is best to run benchmarks in a fixed size heap, using a range of heap sizes to produce a curve that reflects the space-time tradeoff. Using replay compilation and measuring the second iteration of a benchmark is a good way to produce results with low noise.

There is an active debate among memory management and VM researchers about how best to measure performance, and this section is not meant to dictate or advocate any particular position, simply to describe one particular methodology.

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

Perhaps you are not seeing stellar JikesTM RVM performance. If Jikes RVM as described above is not competitive product JVMs, we recommend you test your installation with the DaCapo benchmarks. We expect Jikes RVM performance to be very close to Sun's HotSpot 1.5 server running the DaCapo benchmarks. Of course, running DaCapo well does not guarantee that Jikes RVM runs all codes well.

Some kinds of code will not run fast on Jikes RVM. Known issues include:

1. Jikes RVM start-up may be slow compared to the some product JVMs.
2. Remember that the non-adaptive configurations (`-X:aos:enable_recompilation=false -X:aos:initial_compiler=opt`) 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.
3. Performance on tight loops may suffer. The Jikes RVM mechanism for safe points (thread preemption for garbage collection, on-stack-replacement, profiling, etc) relies on the insertion of a yield 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, or implementing a safe point mechanism that does not require an explicit check.
4. The load balancing in the system is naive and unfair. This can hurt some styles of codes, including bulk-synchronous parallel programs.

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

6.5 Stability of Jikes RVM

Jikes RVM is not as stable as commercial JVMs such as HotSpot or J9. Design your evaluation systems (e.g. scripts) so that they can deal with crashes and deadlocks/livelocks. The latter can be dealt with by running Jikes RVM with a timelimit. For example, if you are using Linux and shell scripts, you can use the `timelimit` program to terminate the Jikes RVM after a set time.

Chapter 7

Modifying Jikes RVM

The sections Coding Style and Coding Conventions give a rough overview on existing coding conventions.

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, refactorings, bug fixes, tests and missing documentation to the project.

7.1 Adding command line options

There are several places in Jikes RVM where options can be defined and each of those uses another approach. There are options for

- MMTk
- VM subsystems such as the compilers and the AOS
- the core VM

7.1.1 MMTk options

Options that are relevant for MMTk can be found in the package `org.mmtk.utility.options`. The only exception are options that must be parsed before the VM can actually start (e.g. `-Xms` and `-Xmx`). Those options are classified as core VM options.

To create a new MMTk option, create a new class in the `org.mmtk.utility.options` package, e.g. `YourOption`. The class must extend a class from the package `org.vmutil.options`. Depending on the type of option, you may want to implement the `validate` method.

YourOption.java

```
/** Your JavaDoc */
public class YourOption extends org.vmutil.options.
    IntOption {

    private static final int DEFAULT_VALUE = 42;
```

```

public YourOption() {
    super(Options.set, "The_name_of_your_option",
        "The_description_of_your_option", DEFAULT_VALUE);
}

@Override
protected void validate() {
    failIf(value < 0 || value > 55,
        "Value_for_your_option_must_be_non-negative_and_
        55_or_smaller");
}
}

```

The newly created option must be linked to the rest of the system. In `Options` from the package `org.mmtk.utility.options`, add a static variable for the option:

Options.java

```
public static YourOption yourOption;
```

Lastly, you will need to create an instance of the option in an appropriate place:

```
Options.yourOption = new YourOption();
```

General options can be created in the constructor of `org.mmtk.plan.Plan`. If the option is specific to a garbage collector, you can create it in one of the collector's classes.

The actual String for the option that will be used on the command line is not determined by MMTk but computed by the VM (Jikes RVM in this case). To look up the chosen key for the newly added option, rebuild Jikes RVM, pass `-X:gc` and examine the output. The output displays a list of MMTk options. This list should now include the newly-added option. To verify that the default value is correctly set, use `-X:gc:printOptions` and check the value.

7.1.2 VM subsystem options

Classes for handling VM subsystem options are generated during the build process. The source files for the options are in `rvm/src-generated/options`. The generated option classes are

- `org.jikesrvm.Options` for VM options
- `org.jikesrvm.adaptive.util.AOSEExternalOptions` for AOS Options
- `org.jikesrvm.compilers.baseline.BaselineOptions` for baseline compiler options
- `org.jikesrvm.compilers.opt.OptOptions` for opt compiler options

Each option class is generated from the matching template file (e.g. `AOS-Options.template` for `AOSEExternalOptions`) and its specific options (e.g. `BooleanOptions.aos.dat` and `ValueOptions.aos.dat`). Additionally, the baseline and opt compiler options also use the shared option files `SharedBooleanOptions.dat` and `SharedValueOptions.dat`.

The existing options and documentation in the files should be sufficient to create new options. Please notify us if the documentation needs to be improved.

7.1.3 Core VM options

Options for the core VM are not specific to a certain subsystem. These options are recognized for all builds of Jikes RVM. This category includes

- bootloader options (e.g. file names for code, data and reference maps of the VM)
- options that must be processed without help of Java code (e.g. `Xms` for the initial heap size or `-Xbootclasspath/p:<cp>` for modification of the boot classpath)
- prefixes for option groups like `-X:opt` or `-X:gc`

Note that an option can be recognized without being available for use. For example, attempting to use `-X:opt` to get help about the optimizing compiler options will fail on builds that don't include the optimizing compiler.

All Core VM options are parsed in the C code. Some are also processed in Java code. If you want to add a core VM option, you will need to modify the bootloader code in order to ensure that options are passed correctly. Recall that the Jikes RVM command line help (`-help`) says the following:

```
Usage: JikesRVM [-options] class [args ...]
           (to execute a class)
or   JikesRVM [-options] -jar jarfile [args ...]
           (to execute a jar file)
```

The bootloader enforces this usage. It parses all options and if it encounters an option that it doesn't recognize, it will assume that this option signifies the start of the application arguments. Consequently, it is necessary to tell the bootloader about all core VM options. The relevant code fragment is shown below.

main.c

```
#define BOOTCLASSPATH_A_INDEX
    BOOTCLASSPATH_P_INDEX + 1
#define PROCESSORS_INDEX
    BOOTCLASSPATH_A_INDEX + 1

#define numNonstandardArgs    PROCESSORS_INDEX + 1

static const char* nonStandardArgs[numNonstandardArgs] =
{
    ... more code here ...
    "-X:availableProcessors=",
};
```

To add a new option, make the following modifications:

main.c

```

#define BOOTCLASSPATH_A_INDEX
    BOOTCLASSPATH_P_INDEX + 1
#define PROCESSORS_INDEX
    BOOTCLASSPATH_A_INDEX + 1
#define YOUR_OPTION_INDEX          PROCESSORS_INDEX +
    1

#define numNonstandardArgs          YOUR_OPTION_INDEX + 1

static const char* nonStandardArgs[numNonstandardArgs] =
{
    ... more code here ...
    "-X:availableProcessors=",
    "-X:your_option=",
};

```

It is recommended that you also add a description of your option to the `non-standardusage` array.

You must modify `processCommandLineArguments(..)` to recognize the option in order to make sure that the option is properly handled and/or passed to the Java code. Failing to do so will cause bugs like RVM-1066. For example, if the option were to take one token, the original code

main.c

```

// All VM directives that take one token
if (STRNEQUAL(token, "-D", 2)
    || STRNEQUAL(token, nonStandardArgs[INDEX], 5)
... more code here ...
    || STRNEQUAL(token, nonStandardArgs[
        BOOTCLASSPATH_P_INDEX], 18)
    || STRNEQUAL(token, nonStandardArgs[
        BOOTCLASSPATH_A_INDEX], 18)
    || STRNEQUAL(token, nonStandardArgs[
        PROCESSORS_INDEX], 14))
{
    CLAs[n_JCLAs++] = token;
    continue;
}

```

would need to be changed to

main.c

```

if (STRNEQUAL(token, "-D", 2)
    || STRNEQUAL(token, nonStandardArgs[INDEX], 5)
... more code here ...
    || STRNEQUAL(token, nonStandardArgs[
        BOOTCLASSPATH_P_INDEX], 18)
    || STRNEQUAL(token, nonStandardArgs[
        BOOTCLASSPATH_A_INDEX], 18)

```

```

        || STRNEQUAL(token , nonStandardArgs [
            PROCESSORS_INDEX] , 14)
        || STRNEQUAL(token , nonStandardArgs [
            YOUR_OPTION_INDEX] , $LENGTH) )
    {
        CLAs[n_JCLAs++] = token ;
        continue ;
    }

```

where `$LENGTH` is the length of the string for your option that you added to the `nonStandardArgs` array.

The above steps are sufficient for options that only need processing in boot-loader code. If your option needs processing in Java, you will need to modify `CommandLineArgs`. Firstly, add a new constant for your option to `PrefixType`. For example, the code

CommandLineArgs.java

```

    BOOTCLASSPATH_ARG,
    BOOTSTRAP_CLASSES_ARG,
    AVAILABLE_PROCESSORS_ARG
}

```

would need to be changed to

CommandLineArgs.java

```

    BOOTCLASSPATH_ARG,
    BOOTSTRAP_CLASSES_ARG,
    AVAILABLE_PROCESSORS_ARG,
    YOUR_OPTION_ARG
}

```

Secondly, you will need to create a new `Prefix` instance in the `prefixes` array. After that, you're ready to add the processing of your option.

7.2 Adding a new garbage collector

7.2.1 Overview

This document describes how to add a new garbage collector to Jikes RVM. We don't address how to design a new GC algorithm, just how to add a "new" GC to the system and then build it. We do this by cloning an existing GC. We leave it to you to design your own GC!

7.2.2 Prerequisites

Ensure that you have got a clean copy of the source (either a recent release or the git HEAD) and can correctly and successfully build one of the base garbage collectors. There's little point in trying to build your own until you can reliably build an existing one. I suggest you start with MarkSweep, and that you use the buildit script:

```
$ bin/buildit <targetmachine> BaseBase MarkSweep
```

Then test your GC:

```
$ bin/buildit <targetmachine> -t gctest BaseBase
MarkSweep
```

You should have seen some output like this:

```
test :
[echo] Test Result for [BaseBaseMarkSweep|gctest]
      InlineAllocation (default) : SUCCESS
[echo] Test Result for [BaseBaseMarkSweep|gctest]
      ReferenceTest (default) : SUCCESS
[echo] Test Result for [BaseBaseMarkSweep|gctest]
      ReferenceStress (default) : SUCCESS
[echo] Test Result for [BaseBaseMarkSweep|gctest]
      FixedLive (default) : SUCCESS
[echo] Test Result for [BaseBaseMarkSweep|gctest]
      LargeAlloc (default) : SUCCESS
[echo] Test Result for [BaseBaseMarkSweep|gctest]
      Exhaust (default) : SUCCESS
```

If this is not working, you should probably go and (re) read the section in the user guide on how to build and run the VM.

7.2.3 Cloning the MarkSweep GC

The best way to do this is in eclipse or a similar tool (see here for how to work with eclipse):

1. Clone the *org.mmtk.plan.marksweep* as *org.mmtk.plan.mygc*
 - (a) You can do this with Eclipse:
 - i. Navigate to *org.mmtk.plan.marksweep* (within *MMTk/src*)
 - ii. Right click over *org.mmtk.plan.marksweep* and select "Copy"
 - iii. Right click again, and select "Paste", and name the target *org.mmtk.plan.mygc* (or whatever you like)
 - iv. This will have cloned the marksweep GC in a new package called *org.mmtk.plan.mygc*
 - (b) or by hand:
 - i. Copy the directory *MMTk/org/mmtk/plan/markswep* to *MMTk/org/mmtk/plan/mygc*
 - ii. Edit each file within *MMTk/org/mmtk/plan/mygc* and change its package declaration to *org.mmtk.plan.mygc*
 - (c) We can leave the GC called "MS" for now (the file names will all be *MMTk/org/mmtk/plan/mygc/MS*.java*)
2. Clone the *BaseBaseMarkSweep.properties* file as *BaseBaseMyGC.properties*:

- (a) Go to build/configs, and right click over BaseBaseMarkSweep.properties, and select "Copy"
- (b) Right click and select "Paste", and paste as BaseBaseMyGC.properties
- (c) Edit BaseBaseMyGC.properties, changing the text:

```
config.mmtk.plan=org.mmtk.plan.markswEEP.MS
```

to

```
config.mmtk.plan=org.mmtk.plan.myGC.MS
```

3. Now test your new GC:

```
$ bin/buildit <targetmachine> -t gctest BaseBase MyGC
```

You should have got similar output to your test of MarkSweep above. That's it. You're done. ☺

7.2.4 Making it Prettier

You may have noticed that when you cloned the package *org.mmtk.plan.markswEEP*, all the classes retained their old names (although in your new namespace; *org.mmtk.plan.myGC*). You can trivially change the class names in an IDE like eclipse. You can do the same with your favorite text editor, but you'll need to be sure that you change the references carefully. To change the class names in eclipse, just follow the procedure below for each class in *org.mmtk.plan.myGC*:

1. Navigate to the class you want changed (eg org.mmtk.plan.myGC.MS)
2. Right click on the class (MS) and select "*Refactor*→*Rename...*" and then type in your new name, (eg *MyGC*)
3. Do the same for each of the other classes:
 - MS → MyGC
 - MSCollector → MyGCCollector
 - MSConstraints → MyGCCConstraints
 - MSMutator → MyGCMutator
 - MSTraceLocal → MyGCTraceLocal
4. Edit your configuration/s to ensure they refer to the renamed classes (since your IDE is unlikely to have done this automatically for you)
 - Go to *build/configs*, and edit each file **MyGC.properties* to refer to your renamed classes

7.2.5 Beyond BaseBaseMyGC

You probably want to build with configurations other than just BaseBase. If so, clone configurations from MarkSweep, just as you did above (for example, clone *FullAdaptiveMarkSweep* as *FullAdaptiveMyGC*). It's best to leave the Fast configurations for last, when you're sure that your GC is working correctly.

7.2.6 What Next?

Once you have this working, you have successfully created and tested your own GC without writing a line of code! You are ready to start the slightly more tricky process of writing your own garbage collector code.

If you are writing a new GC, you should definitely be aware of the MMTk test harness, which allows you to test and debug MMTk in a very well contained pure Java environment, without the rest of Jikes RVM. This allows you to write unit tests and corner cases, and moreover, allows you to edit and debug MMTk entirely from within your IDE.

7.3 Coding Conventions

7.3.1 Assertions in Jikes RVM and MMTk

Partly for historical reasons, we use our own built-in assertion facility rather than the one that appeared in Sun®'s JDK 1.4. All assertion checks have one of the two forms:

```
if (VM.VerifyAssertions) VM._assert(condition)
if (VM.VerifyAssertions) VM._assert(condition, message)
```

`VM.VerifyAssertions` is a public static final field. The `config.assertions` configuration variable determines `VM.VerifyAssertions`' value. If `config.assertions` is set to none, Jikes RVM has no assertion overhead.

If you use the form without a message, then the default message "vm internal error at:" will appear.

If you use the form with a message the message must be a single string literal. Doing string appends in assertions can be a source of horrible performance problems when assertions are enabled (i.e. most development builds). If you want to provide a more detailed error message when the assertion fails, then you must use the following coding pattern:

```
if (VM.VerifyAssertions && condition) {
    ... build message ...
    VM._assert(VM.NOT_REACHED, message);
}
```

An assertion failure is always followed by a stack dump.

Use `VM.ExtremeAssertions` instead of `VM.VerifyAssertions` if the assertion is costly to check but generally useful. These kinds of assertions are only enabled when `config.assertions` is set to `extreme`.

Use `IR.SANITY_CHECK` or `IR.PARANOID` to guard assertions that relate to the intermediate representation in the optimizing compiler.

7.3.2 Assertions in the MMTk Test Harness

The `assert` keyword may be used in the MMTk Harness.

7.3.3 Error Handling

All code in the system needs to detect and handle errors. If you know that your code does not handle certain situations, you should aim to write the code in way that detects these situations. The code also needs to be documented well enough so that users get a hint about the source of the problem. Keep in mind that the Jikes RVM is also used by students who may not be as familiar with the domain as researchers are.

Examples

- The code does not work at all in a certain situation, e.g. it gives incorrect results when the optimizing compiler is enabled or a certain optimization is turned on. In this case, the best approach is to detect the situation and fail fast. This can be done using assertions. You can use `VM.sysFail(..)` for builds without assertions if correct execution after failure is impossible.
- A compiler optimizations fails. The correct approach is to throw an `OptimizingCompilerException` (e.g. via one of the static methods provided by that class). This will lead to a hard failure when `-X:vm:errorsFatal=true` is set (which is the case in regression tests). In other cases, the VM will just revert to using the baseline compiler.
- A command line option has a limited range of values. In MMTk, the correct approach is to implement the `validate()` method for the option. In other places, the value of the option needs to be checked at a suitable time.

7.4 Coding Style

Regrettably, some code in the current system does not follow any consistent coding style. This is an unfortunate residuum of the system's evolution.

We use checkstyle to support a gradually expanding subset of coding conventions. The current set of enforced checkstyle rules are defined by `$RVM_ROOT/build/checkstyle/rvm-checks.xml` and are verified as part of the pre-commit test run. To check for violations of the coding style without running the tests, use `buildit` or run `"ant checkstyle"` from the command line.

7.4.1 File Headers

Every file needs to have the license header.

A Java example of the notices follows.

```
/*
 * This file is part of the Jikes RVM project (http://jikesrvm.org).
 *
 * This file is licensed to You under the Eclipse Public
 * License (EPL);
 * You may not use this file except in compliance with
 * the License. You
```

```

* may obtain a copy of the License at
*
*      http://www.opensource.org/licenses/eclipse-1.0.
*      php
*
* See the COPYRIGHT.txt file distributed with this work
* for information
* regarding copyright ownership.
*/
package org.jikesrvm;

import org.jikesrvm.classloader.ClassLoader; // FILL ME
IN

/**
 * TODO Substitute a brief description of what this
 * program or library does.
 */

```

7.4.2 Coding style description

The Jikes™ RVM coding style guidelines are similar to the Sun® Microsystems "Code Conventions for the Java™ Programming Language", with a few exceptions listed below. Most of the style guide is intuitive; however, please read through the document (or at least look at its sample code).

We have adopted four modifications to the Sun code conventions:

- **Two-space indenting** The Sun coding convention suggests 4 space indenting; however with 80-column lines and four-space indenting, there is very little room left for code. Thus, we recommend using 2 space indenting. There are to be no tabs in the source files or trailing white space on any line.
- **132 column lines in exceptional cases** The Sun coding convention is that lines be no longer than 80 columns. Several Jikes RVM contributors have found this constraining. Therefore, we allow 132 column lines for exceptional cases, such as to avoid bad line breaks.
- **if (VM.VerifyAssertions)** As a special case, the condition `if (VM.VerifyAssertions)` is usually immediately followed by the call to `VM._assert()`, with a single space substituting for the normal newline-and-indentation. See the coding conventions for an example.
- **Capitalized fields** Under the Sun coding conventions, and as specified in The Java Language Specification, Second Edition, the names of fields begin with a lowercase letter. (The only exception they give is for some final static constants, which have names `ALL_IN_CAPITAL_LETTERS`, with underscores separating them.) That convention reserves `IdentifiersBeginningWithACapitalLetterFollowedByMixedCase` for the names of classes and interfaces. However, most of the final fields in the Configuration class

and the Properties interface also are in that format. Since the VM class inherits fields from both Properties and Configuration, that's how we get VM.VerifyAssertions, etc.

7.4.3 Javadoc requirements

All non-trivial files should contain descriptive comments in Javadoc™ form so that documentation can be generated automatically. Of course, additional non-Javadoc source code comments should appear as appropriate.

1. Classes, methods and fields should have a block comment describing them if it makes sense. There is no need to add comments merely for the sake of commenting. For example, it is not necessary to add a comment for a method if the comment does not provide more information than the signature and the method name already do.
2. Javadoc comments must not be copied from methods that are being overridden. If the comment from the method that you are overriding is sufficient, you do not need to provide Javadoc for the newly added method - Javadoc will automatically copy the Javadoc from the overridden method. If you want to extend the comment from the overridden method with new information, use `@inheritDoc` to copy the comment from the superclass and add your text.
3. Javadoc for methods contains a short description of their arguments (using `@param`), the return value (using `@return`) and the exceptions they may throw (using `@throws`).
4. Each class should include `@see` and `@link` references as appropriate.

7.5 Compiler DNA

The Jikes RVM adaptive system uses the compiler DNA found in `org.jikesrvm.adaptive.recompilation.CompilerDNA`. The important values in here are the `compilationRates` and the `speedupRates`. If you modify Jikes RVM then it's likely you need to recalibrate the adaptive system for your changes.

In Jikes RVM 3.1.3 or earlier, do the following:

1. run the compiler-dna test harness ("ant -f test.xml -Dtest-run.name=compiler-dna"). This will automatically compile and run Jikes RVM on SPEC JVM '98. You will want to configure the ant property `external.lib.dir` to be a directory containing your SPEC JVM '98 directory. Your SPEC JVM '98 directory must be named "SPECjvm98".
2. load the xml file `results/tests/compiler-dna/Report.xml` into either an XML viewer (such as a web browser) or into a text editor
3. find the section named `Measure_Compilation_Base`, then look within this section for statistics and find the static `Base.bcb/ms`. For example, `<statistic key="Base.bcb/ms" value="1069.66"/>`. In the `compilationRates` array this will be the value of element 0, it corresponds to how many bytecodes the baseline compiler can compile per millisecond.

4. find the section named `Measure_Compilation_Opt_0` and the statistic `Opt.bcb/ms`. This is element 1 in the `compilationRates` array.
5. find the section named `Measure_Compilation_Opt_1` and the statistic `Opt.bcb/ms`. This is element 2 in the `compilationRates` array.
6. find the section named `Measure_Compilation_Opt_2` and the statistic `Opt.bcb/ms`. This is element 3 in the `compilationRates` array.
7. find the section named `Measure_Performance_Base` and the statistic named `aggregate.best.score` and record its value. For example, for `<statistic key="aggregate.best.score" value="28.90"/>` you would record 28.90.
8. find the section named `Measure_Performance_Opt_0` and the statistic named `aggregate.best.score`. Divide this value by the value you recorded in step 7, this is the value for element 1 in the `speedupRates` array. For example, for `<statistic key="aggregate.best.score" value="137.50"/>` the `speedupRates` array element 1 should have a value of 4.76.
9. find the section named `Measure_Performance_Opt_1` and the statistic named `aggregate.best.score`. As with stage 8 divide this value by the value recorded in step 7, this is the value for element 2 in the `speedupRates` array.
10. find the section named `Measure_Performance_Opt_2` and the statistic named `aggregate.best.score`. As with stage 8 divide this value by the value recorded in step 7, this is the value for element 3 in the `speedupRates` array.

In Jikes RVM 3.1.4 or later, the directory for the test results (e.g. `results/tests/compiler-dna/`) should contain a file `CompilerDNA.xml`. Copy the contents into `CompilerDNA` and modify them so that the code compiles. You should then save `CompilerDNA` and recompile a production RVM which will use these values.

If you are frequently changing the compiler dna, you may want to use the command line option `-X:aos:dna=<file name>` to dynamically load compiler dna data without having to rebuild Jikes RVM.

7.6 Editing Jikes RVM in an IDE

One goal of the JikesRVM project over recent years has been the ability to develop Jikes RVM in a development environment such as Eclipse. This has been possible for the MMTk component since 2005, and as of early 2007 (release 2.9.0) it is possible to work with the majority of the Jikes RVM codebase in Eclipse and similar environments. With Jikes RVM release 2.9.1, setting up your Eclipse environment to work with Jikes RVM became even easier.

7.6.1 Editing JikesRVM in Eclipse

These instructions assume you are working with Jikes RVM version 2.9.1 or later.

1. Create a JikesRVM source tree either via Git checkout or unpacking a distribution.

```
$ git clone https://github.com/JikesRVM/JikesRVM.git
```

2. Create the machine-generated files and eclipse metadata:
3. If you have a recent version of Jikes RVM (3.0 onwards):

```
$ cd jikesrvm
$ bin/buildit --eclipse localhost
```

Note that if you will not or cannot build on your local machine, substitute localhost for the name of a host you can build on (buildit will perform the build remotely and then copy the requisite files back).

4. If you are working on an older version (2.9.1 - 2.9.3), you can follow this procedure:

```
$ cd jikesrvm
$ ant -Dhost.name=ia32-linux -Dconfig.name=
    development
$ ant -Dhost.name=ia32-linux -Dconfig.name=
    development eclipse-project
```

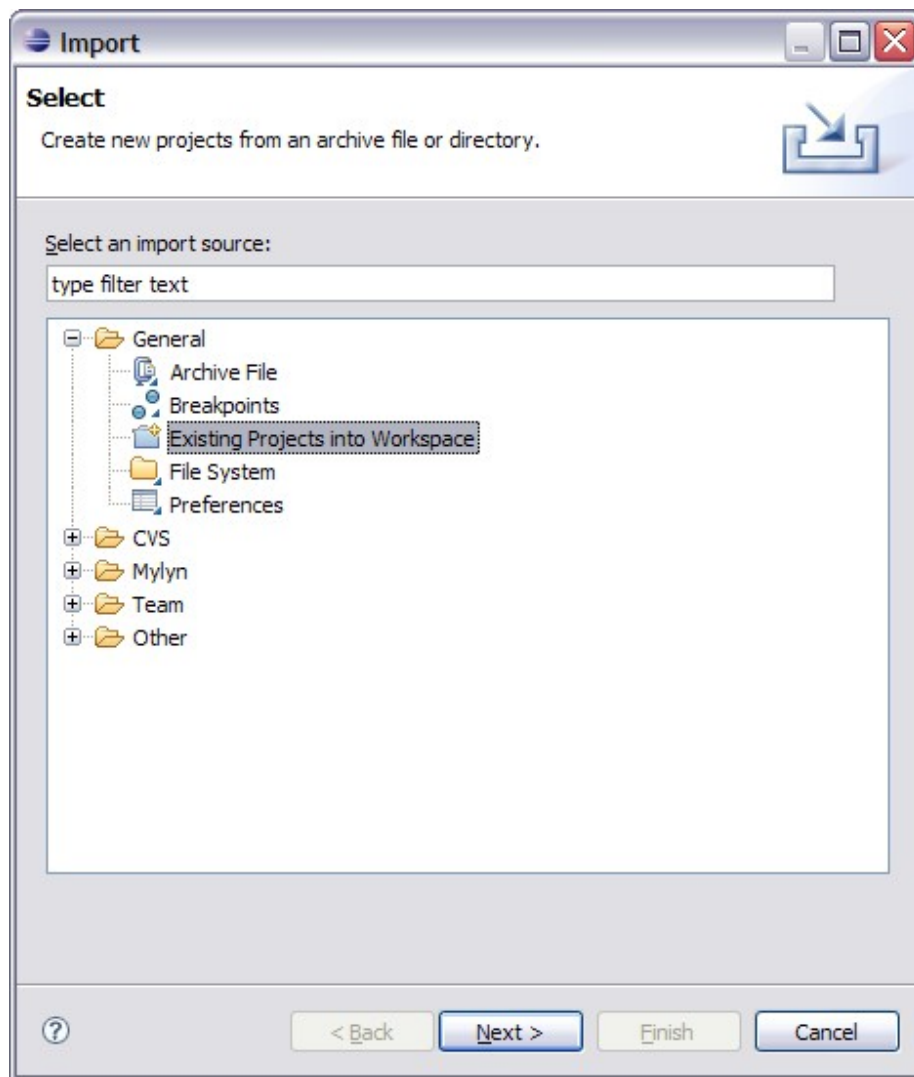
If you will not or cannot build on your local machine:

- (a) copy your tree to build host somehow
 - (b) perform the above ant tasks
 - (c) copy the following generated files and directories back to the machine you will edit on:
 - jikesrvm/.project
 - jikesrvm/.classpath
 - jikesrvm/eclipse
5. Import the newly created Eclipse project into your Eclipse workspace.
 - (a) From Eclipse, select File → Import
 - (b) Select "Existing Projects Into Workspace"
 - (c) Browse to find the top-level directory.
 - (d) Select the project (in this case JikesRVM ia32-linux development)
 - (e) Hit Finish

7.6.2 Setup for easier compliance with the Checkstyle rules

If you consider contributing changes back to Jikes RVM, it is helpful to configure your IDE to comply with the Jikes RVM coding style. The coding style forbids the use of tabs and requires that no line ends with whitespace.

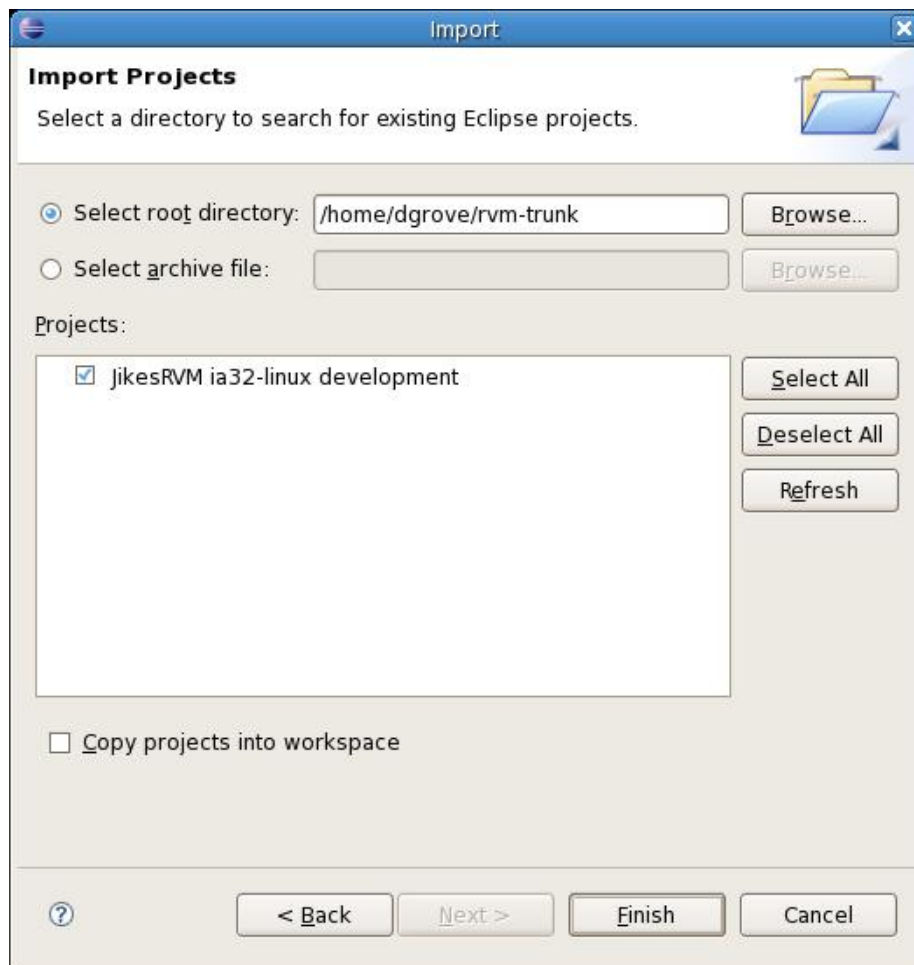
If you have a separate workspace for your work with Jikes RVM, you can set up Eclipse for correct tab usage by configuring the text editors. Go to Window



→ Preferences and then to General → Editors → Text editors (Eclipse 3.6) or Window → Preferences → General → Editors → Text Editors (Eclipse 3.5 and earlier). Check "Insert spaces for tabs". Make sure that "Displayed tab width" is set to 2. This setting affects the non-Java editors (e.g. XML editor for the ant files for the build).

To set the tab width for Java code, you need to setup the Java code style. We currently do not provide a style template, so you will have to define your own. Go to the project properties (e.g. via Project → Properties) and select Java Code Style → Formatter. Check the box "Enable project specific settings" and create a new profile for Jikes RVM. Edit the new profile. In the edit dialog, choose the tab "Indentation". Set the tab policy to "Spaces only" and set both indentation size and tab size to 2. Also make sure that the box "Empty lines" at the bottom of the "Indentation" tab is not checked.

To ensure that you do not introduce whitespace at the end of lines you can



configure Eclipse's Save actions in the project properties at Java Editor → Save Actions. Check the box "Enable project specific settings" and the box "Perform the selected actions on save" as well as "Additional actions". Press "Configure" and check the box "Remove trailing whitespace" in the "Code Organizing" tab.

7.6.3 Editing JikesRVM in NetBeans

1. Follow the instructions for Eclipse including building the eclipse project with ant
2. Install the Eclipse project importer
3. Select File → Import Project → Eclipse Project
 - (a) Choose to import project ignoring project dependencies
 - (b) Select the top-level directory you created with the JikesRVM in as the project to import
 - (c) Select a new folder as the destination (workspace) for the import

(d) Hit Finish

Chapter 8

Profiling Applications with Jikes RVM

The Jikes RVM adaptive system can also be used as a tool for gathering profile data to find application/VM hotspots. In particular, the same low-overhead time-based sampling mechanism that is used to drive recompilation decisions can also be used to produce an aggregate profile of the execution of an application. Here's how.

1. Build an adaptive configuration of Jikes RVM. For the most accurate profile, use the production configuration.
2. Run the application normally, but with the additional command line argument `-X:aos:gather_profile_data=true`
3. When the application terminates, data on which methods and call graph edges were sampled during execution will be printed to stdout (you may want to redirect execution to a file for analysis).

The sampled methods represent compiled versions of methods, so as methods are recompiled and old versions are replaced some of the methods sampled earlier in the run may be OBSOLETE by the time the profile data is printed at the end of the run.

In addition to the sampling-based mechanisms, the baseline compiler can inject code to gather branch probabilities on all executed conditional branches. This profiling is enabled by default in adaptive configurations of Jikes RVM and can be enabled via the command line in non-adaptive configurations (`-X:base:edge_counters=true`). In an adaptive configuration, use `-X:aos:final_report_level=2` to cause the edge counter data to be dumped to a file. In non-adaptive configurations, enabling edge counters implies that the file should be generated (`-X:base:edge_counters=true` is sufficient). The default name of the file is `EdgeCounters`, which can be changed with `-X:base:edge_counter_file=<file_name>`. Note that the profiling is only injected in baseline compiled code, so in a normal adaptive configuration, the gathered probabilities only represent a subset of program execution (branches in opt-compiled code are not profiled). Note that unless the bootimage is (a)

baseline compiled and (b) edge counters were enabled at bootimage writing time, edge counter data will not be gathered for bootimage code.

8.1 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). The code regarding instrumentation lives in the `org.jikesrvm.aos` package.

To instrument all dynamically compiled code, use the following command line arguments to force all dynamically compiled methods to be compiled by the optimizing compiler: `-X:aos:enable_recompilation=false -X:aos:initial_compiler=opt`

8.1.1 Existing Instrumentation Phases

There are several existing instrumentation phases that can be enabled by giving the adaptive optimization system command line arguments. These counters are not synchronized (as discussed later), so they should not be considered precise.

- **Method Invocation Counters** Inserts a counter in each opt compiled method prologue. Prints counters to stderr at end. Enabled by the command line argument `-X:aos:insert_method_counters_opt=true`
- **Yieldpoint Counters** Inserts a counter after each yieldpoint instruction. Maintains a separate counter for backedge and prologue yieldpoints. Enabled by `-X:aos:insert_yieldpoint_counters=true`
- **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.

- **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 `AOSDatabase.debuggingCounterData.getCounterInstructionForEvent(String eventName)`. This method returns an `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 `Inliner.java`. Look for the code guarded by the flag `COUNT_FAILED_METHOD_GUARDS`. Enabled by `-X:aos:insert_debugging_counters=true`

8.1.2 Writing new instrumentation phases

This section 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 `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, `CounterArrayManager`, which implements unsynchronized counters. When instrumentation options are turned on in the adaptive system, `Instrumentation.boot()` creates an instance of a `CounterArrayManager`.

Managed Data: The class `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, `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 `YieldpointCounterData` and `MethodInvocationCounterData`), or one per method. There is also a generic data object called `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 `InsertInstructionCounters.java`, `InsertYieldpointCounters.java`, `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.

Chapter 9

Running Jikes RVM

JikesTM RVM executes Java virtual machine byte code instructions from `.class` files. It does *not* compile JavaTM source code. Therefore, you must compile all Java source files into bytecode using a Java compiler.

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

```
% javac foo.java
% rvm foo
```

The general syntax is

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

You may choose from a myriad of options for the `rvm` command-line. Options fall into two categories: *standard* and *non-standard*. Non-standard options are preceded by `"-X:"` and differ between virtual machines (e.g. Jikes RVM's options may not be available in HotSpot and vice-versa).

9.1 Standard Command-Line Options

We currently support a subset of the JDK 1.5 standard options. Below is a list of all options and their descriptions. Unless otherwise noted each option is supported in Jikes RVM.

9.2 Non-Standard Command-Line Options

It is generally the case that the non-standard options may change from one release to another. However, the core and memory non-standard options that are listed here don't change often.

The bulk of the non-standard options are grouped according to the subsystem that they control. See below for details.

Option	Description
-cp or -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 the run
-showversion	print current VM version and continue running
-fullversion	like "-version", but with more information
-? or -help	print help message
-X	print help on non-standard options
-jar	execute a jar file
-javaagent:<jarpath>[=<options>]	load Java programming language agent, see <code>java.lang.instrument</code>

9.2.1 Core Non-Standard Command-Line Options

Option	Description
-X:verbose	Print out additional lowlevel information for GC and hardware trap handling
-X:verboseBoot=<number>	Print out additional information while VM is booting, using verbosity level <number>
-X:sysLogfile=<filename>	Write standard error message to <filename>
-X:ic=<filename>	Read boot image code from <filename>
-X:id=<filename>	Read boot image data from <filename>
-X:ir=<filename>	Read boot image ref map from <filename>
-X:vmClasses=<path>	Load the <code>org.jikesrvm.*</code> and <code>java.*</code> classes from <path>
-X:processors=<number "all">	The number of processors that the garbage collector will use

9.2.2 Memory Non-Standard Command-Line Options

9.2.3 Subsystem Non-Standard Command-Line Options

The other non-standard command line options are not listed here because they will change from time to time. To get a current list of these options, you can do the following:

- To find out which subsystems are available in the image you're using:

```
rvm -X
```

Option	Description
<code>-Xms<number><unit></code>	Initial size of heap where <code><number></code> is an integer, an extended-precision floating point or a hexadecimal value and <code><unit></code> is one of T (Terabytes), G (Gigabytes), M (Megabytes), pages (of size 4096), K (Kilobytes) or <code><no unit></code> for bytes
<code>-Xmx<number><unit></code>	Maximum size of heap. See above for definition of <code><number></code> and <code><unit></code>

- To get information about the options for a specific subsystem (e.g. the garbage collection subsystem)

```
rvm -X:gc:help
```

- To find out the current values of the options for a specific subsystem (e.g. the optimizing compiler subsystem)

```
rvm -X:opt:printOptions
```

9.3 Running Jikes RVM with Valgrind

Jikes RVM can run under valgrind, as of 29-Aug-2007. Applying a patch of this revision to release 3.2.1 should also produce a working system. Versions of valgrind CVS prior to release 3.0 are also known to have worked. The current git HEAD should also work with Valgrind.

To run a Jikes RVM build with valgrind, use the `-wrap` flag to invoke valgrind, eg

```
rvm -wrap "path/to/valgrind --smc-check=all <valgrind -
options>" <jikesrvm-options> ...
```

Note that the full path to valgrind must be specified.

This will insert the invocation of valgrind at the appropriate place for it to operate on Jikes RVM proper rather than a wrapper script.

Under some circumstances, valgrind will load shared object libraries or allocate memory in areas of the heap that conflict with Jikes RVM. Using the flag `-X:gc:eagerMmapSpaces=true` will prevent and/or detect this. If this flag reveals errors while mapping the spaces, you will need to rearrange the heap to avoid the addresses that Valgrind is occupying.

If you're using a recent Jikes RVM version (later than 3.1.3), you may want to change the line

```
private static final boolean
ALLOW_READS_OF_UNDEFINED_MEMORY = true;
```

in `org.jikesrvm.jni.JNIGenericHelpers` to

```
private static final boolean  
    ALLOW_READS_OF_UNDEFINED_MEMORY = false;
```

to prevent false positives from Valgrind.

Chapter 10

Testing Jikes RVM

Jikes RVM includes provisions to run unit tests as well as functional and performance tests. It also includes a number of actual tests, both unit and functional ones.

10.1 Unit Tests

Jikes RVM makes writing simple unit tests easy. Simply give your JUnit 4 tests a name ending in `Test` and place test sources under `rvm/test-src`. The tests will be picked up automatically.

The tests are then run on the bootstrap VM, i.e. the JVM used to build Jikes RVM. You can also configure the build to run unit tests on the newly built Jikes RVM. Note that this may significantly increase the build times of slow configurations (e.g. `prototype` and `prototype-opt`).

If you are developing new unit tests, it may be helpful to run them on an existing Jikes RVM image. This can be done by using the Ant target `unit-tests-on-existing-image`. The path for the image is determined by the usual properties of the Ant build.

10.2 Functional and Performance Tests

See External Test Resources for details or downloading prerequisites for the functional tests. The tests are executed using an Ant build file and produce results that conform to the definition below. The results are aggregated and processed to produce a high level report defining the status of Jikes RVM.

The testing framework was designed to support continuous and periodical execution of tests. A *"test-run"* occurs every time the testing framework is invoked. Every *"test-run"* will execute one or more *"test-configuration"*s. A *"test-configuration"* defines a particular build *"configuration"* (See Configuring Jikes RVM for details) combined with a set of parameters that are passed to Jikes RVM during the execution of the tests. i.e. a particular *"test-configuration"* may pass parameters such as `-X:aos:enable_recompilation=false` `-X:aos:initial_compiler=opt` `-X:irc:01` to test the Level 1 Opt compiler optimizations.

Every *"test-configuration"* will execute one or more *"group"*s of tests. Every *"group"* is defined by a Ant build.xml file in a separate sub-directory of

`$RVM_ROOT/testing/tests`. Each *test* has a number of input parameters such as the classname to execute, the parameters to pass to Jikes RVM or to the program. The *test* records a number of values such as execution time, exit code, result, standard output etc. and may also record a number of statistics if it is a performance test.

The project includes several different types of test runs and the description of each the test runs and their purpose is given in Test Run Descriptions.

Note that the `buildit` script provides a fast and easy way to build and the system. The script is simply a wrapper around the mechanisms described below.

10.2.1 Ant properties

There is a number of ant properties that control the test process. Besides the properties that are already defined in Building Jikes RVM, special test properties may also be specified.

10.2.2 Defining a test-run

A *test-run* is defined by a number of properties located in a property file located in the `build/test-runs/` directory.

The property `test.configs` is a whitespace separated list of test-configuration "tags". Every tag uniquely identifies a particular test-configuration. Every test-configuration is defined by a number of properties in the property file that are prefixed with `test.config.<tag>`. See the test run property table for the possible properties.

The order of the test-configurations in *test.configs* is the order that the test-configurations are tested. The order of the groups in *test.config.<tag>.test* is the order that the tests are executed.

The simplest test-run is defined in the following figure. It will use the build configuration *prototype* and execute tests in the *basic* group.

build/test-runs/simple.properties

```
test.configs=prototype
test.config.prototype.tests=basic
```

The test process also expands properties in the property file so it is possible to define a set of tests once but use them in multiple test-configurations as occurs in the following figure. The groups *basic*, *optests* and *dacapo* are executed in both the *prototype* and *prototype-opt* test configurations.

build/test-runs/property-expansion.properties

```
test.set=basic optests dacapo
test.configs=prototype prototype-opt
test.config.prototype.tests=${test.set}
test.config.prototype-opt.tests=${test.set}
```

Each test can have additional parameters specified that will be used by the test infrastructure when starting the Jikes RVM instance to execute the test. These additional parameters are described in table for test specific parameters.

Parameter	Description	Default Property	Default Value
initial.heapsize	The initial size of the heap.	<code>\${test.initial.heapsize}</code>	<code>\${config.default-heapsize.initial}</code>
max.heapsize	The initial size of the heap.	<code>\${test.max.heapsize}</code>	<code>\${config.default-heapsize.maximum}</code>
max.opt.level	The maximum optimization level for the tests or an empty string to use the Jikes RVM default.	<code>\${test.max.opt.level}</code>	<code>""</code>
processors	The number of processors to use for garbage collection for the test or 'all' to use all available processors.	<code>\${test.processors}</code>	all
time.limit	The time limit for the test in seconds. After the time limit expires the Jikes RVM instance will be forcefully terminated.	<code>\${test.time.limit}</code>	1000
class.path	The class path for the test.	<code>\${test.class.path}</code>	
extra.args	Extra arguments that are passed to Jikes RVM.	<code>\${test.rvm.extra.args}</code>	<code>""</code>
exclude	If set to true, the test will be not be executed.		<code>""</code>

Table 10.3: Test specific parameters

To determine the value of a test specific parameters, the following mechanism is used:

1. Search for one of the the following ant properties, in order.
 - (a) `test.config.<build-configuration>.<group>.<test>.<parameter>`
 - (b) `test.config.<build-configuration>.<group>.<parameter>`
 - (c) `test.config.<build-configuration>.<parameter>`

- (d) `test.config.<build-configuration>.<group>.<test>.<parameter>`
 - (e) `test.config.<build-configuration>.<group>.<parameter>`
2. If none of the above properties are defined then use the parameter that was passed to the `<rvm>` macro in the ant build file.
 3. If no parameter was passed to the `<rvm>` macro then use the default value which is stored in the "Default Property" as specified in the above table. By default the value of the "Default Property" is specified as the "Default Value" in the above table, however a particular build file may specify a different "Default Value".

10.2.3 Excluding tests

Sometimes it is desirable to exclude tests. The test exclusion may occur as the test is known to fail on a particular target platform, build configuration or maybe it just takes too long. To exclude a test, you must define the test specific parameter "exclude" to true either in `.ant.properties` or in the test-run properties file.

For example, at the time of writing the Jikes RVM does not fully support volatile fields and as a result the test named "TestVolatile" in the "basic" group will always fail. Rather than being notified of this failure we can disable the test by adding a property such as `"test.config.basic.TestVolatile.exclude=true"` into test-run properties file.

10.2.4 Executing a test-run

The tests are executed by the Ant driver script `test.xml`. The `test-run.name` property defines the particular test-run to execute and if not set defaults to `"sanity"`. The command `ant -f test.xml -Dtest-run.name=simple` executes the test-run defined in `build/test-runs/simple.properties`. When this command completes you can point your browser at `${results.dir}/tests/${test-run.name}/Report.html` to get an overview on test run or at `${results.dir}/tests/${test-run.name}/Report.xml` for an XML document describing test results.

10.2.5 Jenkins integration

Executing a test-run on a recent version of Jikes RVM (later than 3.1.3) also produces a file called `MinimalReport-JUnitFormat.xml`. This file contains the test results in a format that's suitable for processing as JUnit results in Jenkins. To use this file in Jenkins, select the appropriate job and add a post-build step to publish JUnit test result reports. Ensure that the box "Retain long standard output/error" is *not checked*. If long standard output/error is retained, you will likely run into `OutOfMemoryErrors` when Jenkins parses the results of larger test runs. For example, parsing results from the sanity test run (which contains >3000 test cases) exhausts 5 GB heaps.

10.3 External Test Resources

The tests included in the source tree are designed to test the correctness and performance of the Jikes RVM. This document gives a step by step instructions for setting up the external dependencies for these tests.

The first step is selecting the base directory where all the external code is to be located. The property `external.lib.dir` needs to be set to this location. i.e.

```
> echo "external.lib.dir=/home/peter/Research/External"
>> .ant.properties
> mkdir -p /home/peter/Research/External
```

Then you need to follow the instructions below for the desired benchmarks. The instructions assume that the environment variable `BENCHMARK_ROOT` is set to the same location as the `external.lib.dir` property.

10.3.1 Open Source Benchmarks

In the future other benchmarks such as BigInteger, Ashes or Volano may be included.

Dacapo

Dacapo describes itself as "This benchmark suite is intended as a tool for Java benchmarking by the programming language, memory management and computer architecture communities. It consists of a set of open source, real world applications with non-trivial memory loads. The suite is the culmination of over five years work at eight institutions, as part of the DaCapo research project, which was funded by a National Science Foundation ITR Grant, CCR-0085792." The release needs to be downloaded and placed in the `$BENCHMARK_ROOT/-dacapo/` directory. i.e.

```
> mkdir -p $BENCHMARK_ROOT/dacapo/
> cd $BENCHMARK_ROOT/dacapo/
> wget http://sourceforge.net/projects/dacapobench/files/
  archive/2006-10/dacapo-2006-10.jar/download?use_mirror
  =autoselect"
```

jBYTEmark

jBYTEmark was a benchmark developed by Byte.com a long time ago.

```
> mkdir -p $BENCHMARK_ROOT/jBYTEmark-0.9
> cd $BENCHMARK_ROOT/jBYTEmark-0.9
> wget http://img.byte.com/byte/bmark/jbyte.zip
> unzip -jo jbyte.zip 'app/class/*'
> unzip -jo jbyte.zip 'app/src/jBYTEmark.java'
> ... Edit jBYTEmark.java to delete "while (true) {}" at
  the end of main. ...
> javac jBYTEmark.java
> jar cf jBYTEmark-0.9.jar *.class
> rm -f *.class jBYTEmark.java
```

CaffeineMark

CaffeineMark describes itself as "The CaffeineMark is a series of tests that measure the speed of Java programs running in various hardware and software configurations. CaffeineMark scores roughly correlate with the number of Java instructions executed per second, and do not depend significantly on the the amount of memory in the system or on the speed of a computers disk drives or internet connection."

```
> mkdir -p $BENCHMARKROOT/CaffeineMark-3.0
> cd $BENCHMARKROOT/CaffeineMark-3.0
> wget http://www.benchmarkhq.ru/cm30/cmkit.zip
> unzip cmkit.zip
```

xerces

Process some large documents using xerces XML parser.

```
> cd $BENCHMARKROOT
> wget http://archive.apache.org/dist/xml/xerces-j/Xerces-
  -J-bin.2.8.1.tar.gz
> tar xzf Xerces-J-bin.2.8.1.tar.gz
> mkdir -p $BENCHMARKROOT/xmlFiles
> cd $BENCHMARKROOT/xmlFiles
> wget http://www.ibiblio.org/pub/sun-info/standards/xml/
  eg/shakespeare.1.10.xml.zip
> unzip shakespeare.1.10.xml.zip
```

Soot

Soot describes itself as "Originally, Soot started off as a Java optimization framework. By now, researchers and practitioners from around the world use Soot to analyze, instrument, optimize and visualize Java and Android applications."

```
> mkdir -p $BENCHMARKROOT/soot-2.2.3
> cd $BENCHMARKROOT/soot-2.2.3
> wget http://www.sable.mcgill.ca/software/sootclasses-
  -2.2.3.jar
> wget http://www.sable.mcgill.ca/software/jasminclasses-
  -2.2.3.jar
```

Java Grande Forum Sequential Benchmarks

Java Grande Forum Sequential Benchmarks is a benchmark suite designed for single processor execution.

```
> mkdir -p $BENCHMARKROOT/JavaGrandeForum
> cd $BENCHMARKROOT/JavaGrandeForum
> wget http://www2.epcc.ed.ac.uk/javagrande/seq/jgf_v2.
  tar.gz
> tar xzf jgf_v2.tar.gz
```

Java Grande Forum Multi-threaded Benchmarks

Java Grande Forum Multi-threaded Benchmarks is a benchmark suite designed for parallel execution on shared memory multiprocessors.

```
> mkdir -p $BENCHMARKROOT/JavaGrandeForum
> cd $BENCHMARKROOT/JavaGrandeForum
> wget http://www2.epcc.ed.ac.uk/javagrande/threads/
    jgf_threadv1.0.tar.gz
> tar xzf jgf_threadv1.0.tar.gz
```

JLex Benchmark

JLex is a lexical analyzer generator, written for Java, in Java.

```
> mkdir -p $BENCHMARKROOT/JLex-1.2.6/classes/JLex
> cd $BENCHMARKROOT/JLex-1.2.6/classes/JLex
> wget http://www.cs.princeton.edu/~appel/modern/java/
    JLex/Archive/1.2.6/Main.java
> mkdir -p $BENCHMARKROOT/QBJC
> cd $BENCHMARKROOT/QBJC
> wget http://www.ocf.berkeley.edu/~horie/qbjlex.txt
> mv qbjlex.txt qb1.lex
```

10.3.2 Proprietary Benchmarks

SPECjbb2005

SPECjbb2005 describes itself as "SPECjbb2005 (Java Server Benchmark) is SPEC's benchmark for evaluating the performance of server side Java. Like its predecessor, SPECjbb2000, SPECjbb2005 evaluates the performance of server side Java by emulating a three-tier client/server system (with emphasis on the middle tier). The benchmark exercises the implementations of the JVM (Java Virtual Machine), JIT (Just-In-Time) compiler, garbage collection, threads and some aspects of the operating system. It also measures the performance of CPUs, caches, memory hierarchy and the scalability of shared memory processors (SMPs). SPECjbb2005 provides a new enhanced workload, implemented in a more object-oriented manner to reflect how real-world applications are designed and introduces new features such as XML processing and BigDecimal computations to make the benchmark a more realistic reflection of today's applications." SPECjbb2005 requires a license to download and use.

SPECjbb2005 can be run on command line via:

```
$RVMROOT/rvm -X:processors=1 -Xms400m -Xmx600m -
    classpath jbb.jar:check.jar spec.jbb.JBBmain -propfile
    SPECjbb.props
```

SPECjbb2005 may also be run as part regression tests.

```
> mkdir -p $BENCHMARKROOT/SPECjbb2005
> cd $BENCHMARKROOT/SPECjbb2005
> ... Extract package here ???...
```

SPECjbb2000

SPECjbb2000 describes itself as "SPECjbb2000 (Java Business Benchmark) is SPEC's first benchmark for evaluating the performance of server-side Java. Joining the client-side SPECjvm98, SPECjbb2000 continues the SPEC tradition of giving Java users the most objective and representative benchmark for measuring a system's ability to run Java applications." SPECjbb2000 requires a license to download and use. Benchmarks should no longer be performed using SPECjbb2000 as the benchmarks have very different characteristics.

```
> mkdir -p $BENCHMARK_ROOT/SPECjbb2000
> cd $BENCHMARK_ROOT/SPECjbb2000
> ... Extract package here ???...
```

SPEC JVM98 Benchmarks

JVM98 features: "Measures performance of Java Virtual Machines. Applicable to networked and standalone Java client computers, either with disk (e.g., PC, workstation) or without disk (e.g., network computer) executing programs in an ordinary Java platform environment. Requires Java Virtual Machine compatible with JDK 1.1 API, or later." SPEC JVM98 Benchmarks require a license to download and use.

```
> mkdir -p $BENCHMARK_ROOT/SPECjvm98
> cd $BENCHMARK_ROOT/SPECjvm98
> ... Extract package here ???...
```

10.4 Test Run Descriptions

The Jikes RVM project contains several different test runs with different purposes. This document attempts to capture the purpose of each different test run.

10.4.1 pre-commit

This test run **MUST** be run prior to committing code. It is relatively short and is designed to capture as many potential bugs in the shortest possible time. It is expected that the pre-commit test run will take 7-15 minutes on modern intel architecture.

10.4.2 core

There is a set of workloads we consider important (i.e. dacapo and SPEC*). There is a set of build configurations we consider important (ie prototype, development, production). We as a group wish to guarantee that all important workloads will run correctly on all important build configurations, i.e. We should **NEVER** regress. The core test run is designed to identify as early as possible any failures in this matrix of build configuration x workload. It is run continuously 24 hours a day (or at least every time a change is made). It is

expected that the core test run will take 2-6 hours to complete depending on the environment.

The best way to identify the failures is to stress test the system by forcing frequent garbage collections and compilation at specific optimization levels (and perhaps frequent thread switching and frequent OSR events in the future). It is critical that we have a stable research base so intermittent failures are NOT acceptable. If we can not pass a stress test then there is no guarantee that we have a stable research base.

10.4.3 sanity

The sanity test runs cover a larger number of build configurations and workloads. They may not always pass and may test many of the less frequently used configurations (gctrace, gcspy, and individual stress tests) and less important workloads. Performance tests are also included in this test run. Something we use to gauge the health of the project as a whole and to track regressions. These are run once a day on major platforms. These time to complete can vary but expected to take several hours at the least.

10.4.4 Other test runs

A set of test runs that are used for testing specific aspects of the system from performance, gcmap bug finding, io hammering etc. There may also be a set of personal/site-specific test runs included in this set that are not checked into Git repository.

10.4.5 Summary

We must NEVER regress in the core test run. The pre-commit test run attempts to ensure no core regressions this while keeping running time reasonable. The sanity test run gives us an overall picture on the health of the code base. While the other test runs are used at different times for different purposes.

Property	Description	Default
test-run.name	The name of the test-run. The name should match one of the files located in the build/test-runs/ directory minus the '.properties' extension.	pre-commit
results.dir	The directory where Ant stores the results of the test run.	\${jikesrvm.dir}/results
results.archive	The directory where Ant gzips and archives a copy of test run results and reports.	\${results.dir}/archive
send.reports	Define this property to send reports via email.	(Undefined)
mail.from	The from address used when emailing report.	jikesrvm-core@lists.sourceforge.net
mail.to	The to address used when emailing report.	jikesrvm-regression@lists.sourceforge.net
mail.host	The host to connect to when sending mail.	localhost
mail.port	The port to connect to when sending mail.	25
<configuration>.built	If set to true, the test process will skip the build step for specified configurations. For the test process to work the build must already be present.	(Undefined)
skip.build	If defined the test process will skip the build step for all configurations and the javadoc generation step. For the test process to work the build must already be present.	(Undefined)
skip.javadoc	If defined the test process will skip the javadoc generation step.	(Undefined)

Table 10.1: Test properties

Property	Description	Default
tests	The names of the test groups to execute.	None
name	The unique identifier for test-configuration.	""
configuration	The name of the Jikes RVM build configuration to test.	<tag>
target	The name of the Jikes RVM build target. This can be used to trigger compilation of a profiled image	"main"
mode	The test mode. May modify the way test groups execute. See individual groups for details.	""
extra.rvm.args	Extra arguments that are passed to the Jikes RVM. These may be varied for different runs using the same image.	""

Table 10.2: Test run properties

Chapter 11

The MMTk Test Harness

11.1 Overview

The MMTk harness is a debugging tool. It allows you to run MMTk with a simple client - a simple Java-like scripting language - which can explicitly allocate objects, create and delete references, etc. This allows MMTk to be run and debugged stand-alone, without the entire VM, greatly simplifying initial debugging and reducing the edit-debug turnaround time. This is all accessible through the command line or an IDE such as eclipse.

The harness can be run standalone or via Eclipse (or other IDE).

11.2 Standalone

```
ant mmtk-harness
java -jar target/mmtk/mmtk-harness.jar <script-file> [
    options...]
```

There is a collection of sample scripts in the `MMTk/harness/test-scripts` directory. There is a simple wrapper script that runs all the available scripts against all the collectors,

```
bin/test-mmtk [options...]
```

This script prints a PASS/FAIL line as it goes, and puts detailed output in `results/mmtk`.

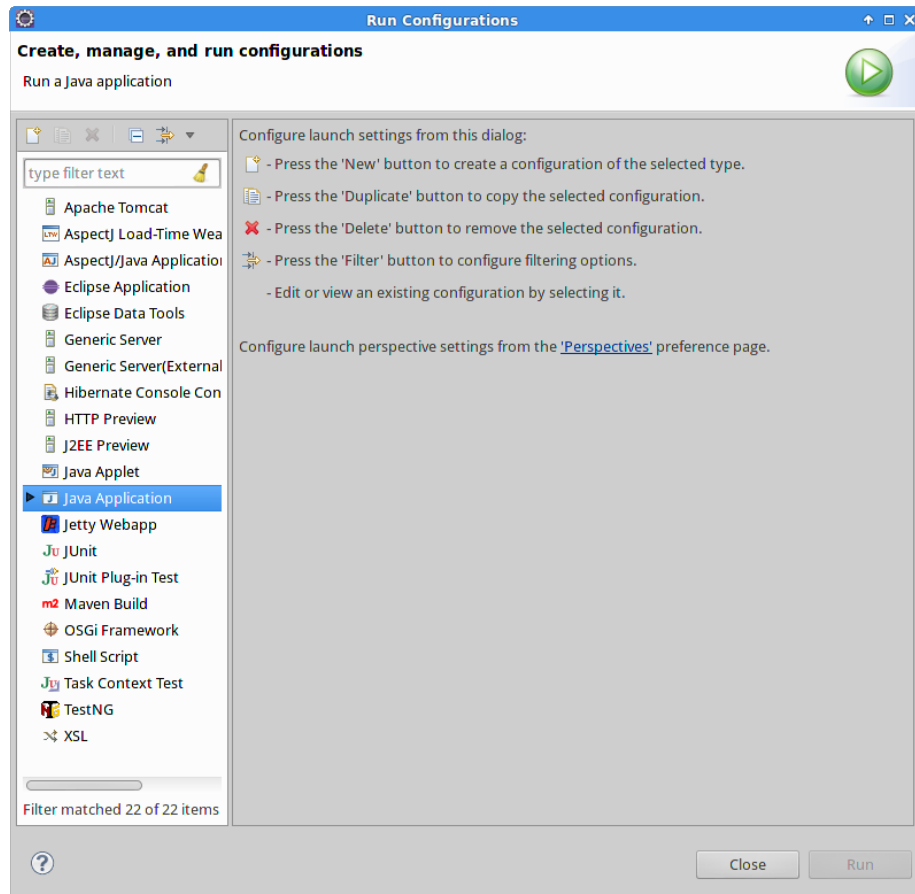
11.3 In Eclipse

```
bin/buildit localhost --mmtk-eclipse
```

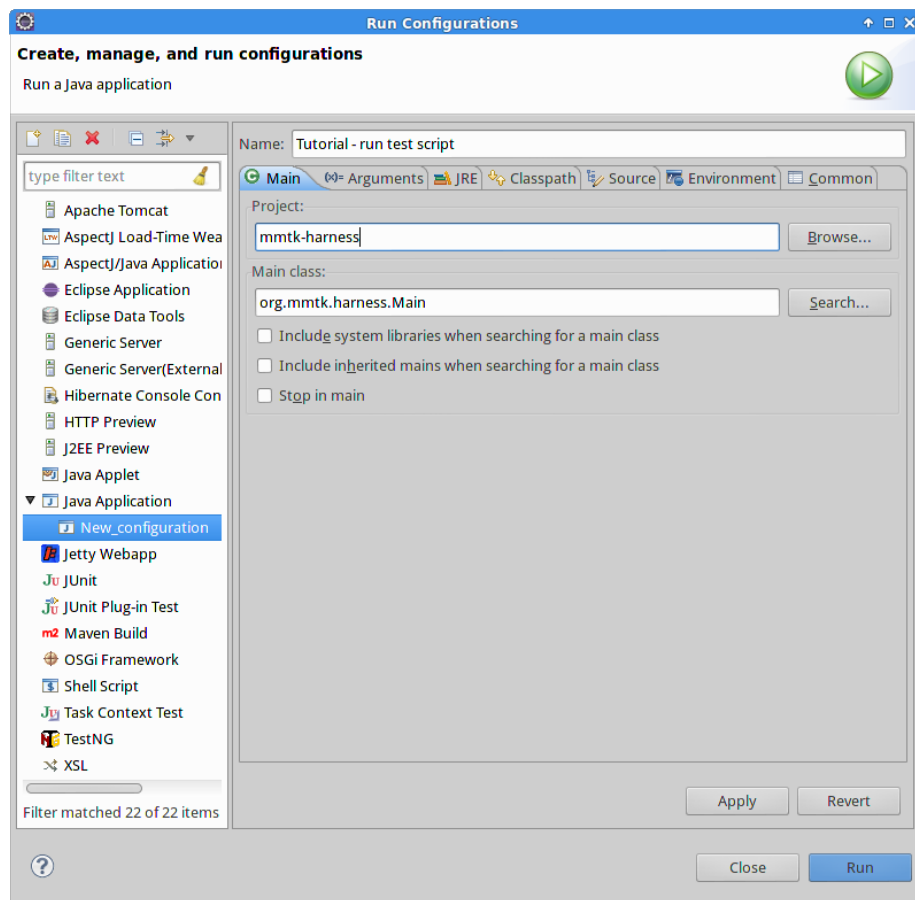
Or in versions before 3.1.1

```
ant mmtk-harness && ant mmtk-harness-eclipse-project
```

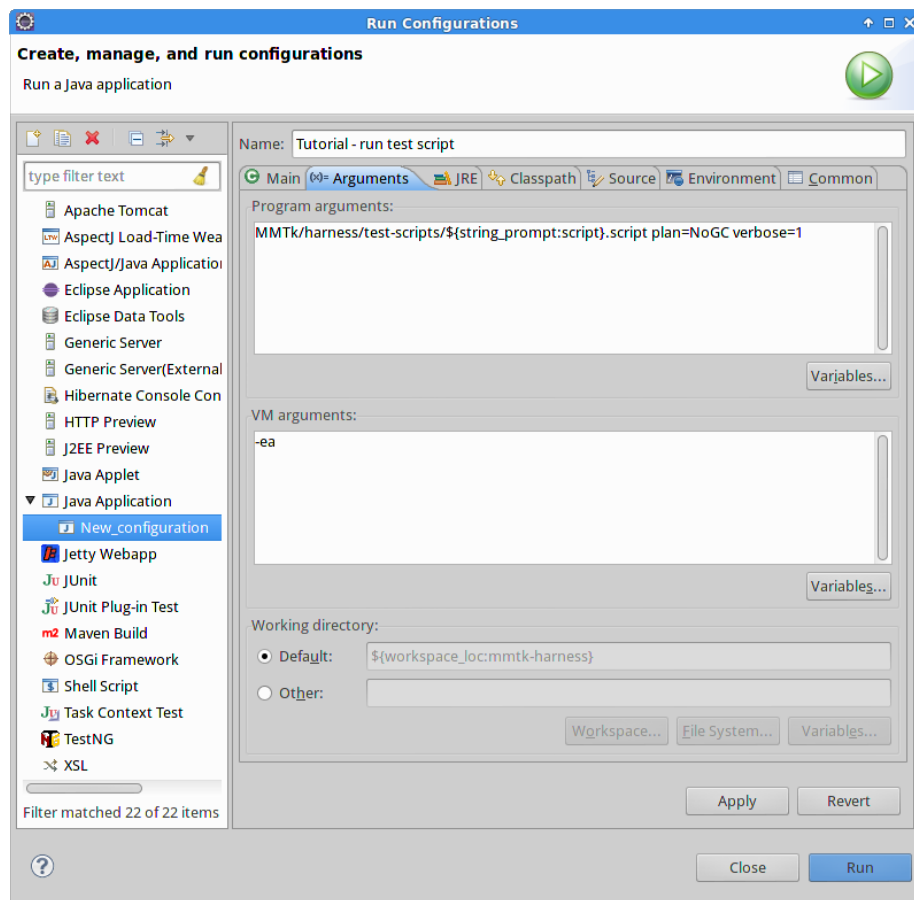
Refresh the project (or import it into eclipse), and then run 'Project → Clean'. Define a new run configuration with main class `org.mmtk.harness.Main`. Click Run (actually the down-arrow next to the the green button), choose 'Run Configurations...'



Select "Java Application" from the left-hand panel, and click the "new" icon (top left). Fill out the Main tab as below



Fill out the Arguments tab as below



The harness makes extensive use of the java 'assert' keyword, so you should run the harness with '-ea' in the VM options.

Click 'Apply' and then 'Run' to test the configuration. Eclipse will prompt for a value for the 'script' variable - enter the name of one of the available test scripts, such as 'Lists', and click OK. The scripts provided with MMTk are in the directory `MMTk/harness/test-scripts`.

You can configure eclipse to display vmmagic values (Address/ObjectReference/etc) using their toString method through the Eclipse → Preferences... → Java → Debug → Detail Formatters menu. The simplest option is to check the box to use toString 'As the label for all variables'.

11.4 Test harness options

Options are passed to the test harness as 'keyword=value' pairs. The standard MMTk options that are available through JikesRVM are accepted (leave off the "-X:gc:"), as well as the following harness-specific options:

Option	Meaning	Defaults to
plan	The MMTk plan class.	org.mmtk.plan.marksweep.MS

initHeap	Initial heap size. It is also a good idea to use 'variableSizeHeap=false', since the heap growth manager uses elapsed time to make its decisions, and time is seriously dilated by the MMTk Harness.
maxHeap	Maximum heap size (default: 64 pages)
trace	<p>Debugging messages from the MMTk Harness. Useful trace options include</p> <ul style="list-style-type: none"> • ALLOC - trace object allocation • AVBYTE - Mutations of the 'available byte' in each object header • COLLECT - Detailed information during GC • HASH - Hash code operations • MEMORY - page-level memory operations (map, unmap, zero) • OBJECT - trace object mutation events • REFERENCES - Reference type processing • REMSET - Remembered set processing • SANITY - Gives detailed information during Harness sanity checking • TRACEOBJECT - Traces every call to traceObject during GC (requires MMTk support) <p>See the class <code>org.mmtk.harness.lang.Trace</code> for more details and trace options - most of the remaining options are only of interest to maintainers of the Harness itself.</p>
watchAddress	Set a watchpoint on a given address or comma-separated list of addresses. The harness will display every load and store to that address.
watchObject	Watch modifications to a given object or comma-separated list of objects, identified by object ID (sequence number).
gcEvery	<p>Force frequent GCs. Options are</p> <ul style="list-style-type: none"> • ALLOC - GC after every object allocation • SAFEPOINT - GC at every GC safepoint

scheduler	<p>Optionally use the deterministic scheduler. Options are</p> <ul style="list-style-type: none"> • JAVA (default) - Threads in the script are Java threads, scheduled by the host JVM • DETERMINISTIC - Threads are scheduled deterministically, with yield points at every memory access.
schedulerPolicy	<p>Select from several scheduling policies,</p> <ul style="list-style-type: none"> • FIXED - Threads yield every 'nth' yield point • RANDOM - Threads yield according to a pseudo-random policy • NEVER - Threads only yield at mandatory yieldpoints
yieldInterval	For the FIXED scheduling policy, the yield frequency.
randomPolicyLength	Parameters for the RANDOM scheduler policy. Whenever a thread is created, the scheduler fixes a yield pattern of 'length' integers between 'min' and 'max'. These numbers are used as yield intervals in a circular manner.
randomPolicySeed	
randomPolicyMin	
randomPolicyMax	
policyStats	Dump statistics for the deterministic scheduler's yield policy.
bits=32 64	Select between 32 and 64-bit memory models.
dumpPcode	Dump the pseudo-code generated by the harness interpreter
timeout	Abort collection if a GC takes longer than this value (seconds). Defaults to 30.

11.5 Scripts

The `MMTk/harness/test-scripts` directory contains several test scripts.

11.6 Scripting language

11.6.1 Basics

The language has three types: integer, object and user-defined. The object type behaves essentially like a double array of pointers and integers (odd, I know, but the scripting language is basically concerned with filling up the heap with objects of a certain size and reachability). User-defined types are like Java objects without methods, 'C' structs, Pascal record types etc.

Script	Purpose	Description
Alignment	Test allocator alignment behaviour	Tests alignment by creating a list of objects aligned to a mixture of 4-byte and 8-byte boundaries.
CyclicGarbage	Test cycle detector in Reference Counting collectors	Creates large amounts of cyclic garbage in the form of circular linked lists.
FixedLive	General collection test	Harness version of the FixedLive GC micro-benchmark. Creates a binary tree, then allocates short-lived objects to force garbage collections.
HashCode	Hash code test.	Creates objects and verifies that their hashcode is unchanged after a GC.
LargeObject	Large object allocator test	Creates objects with sizes ranging from 2 to 32 pages (8k to 128k bytes).
Lists	Generational collector stress test	Creates a set of lists of varying lengths, and then allocates to force collections. Ensures that there are Mature→Nursery, Nursery→Mature and Stack→Nursery and Stack→Mature pointers at every GC. Remsets get a serious workout.
OutOfMemory	Tests out-of-memory handling.	Allocates a linked list that grows until the heap fills up.
Quicksort	General collection test	Implements a list-based quicksort.
ReferenceTypes	Reference type test	Creates Weak references, forces collections and ensures that they are correctly handled.
Spawn	Concurrency test	Creates lots of threads which allocate objects.
SpreadAlloc	Free-list allocator test	Creates large numbers of objects with random size distributions, keeping a fraction of the objects alive.
SpreadAlloc16	Concurrent free-list allocator test	A multithreaded version of SpreadAlloc.

Objects and user-defined types are allocated with the 'alloc' statement: `alloc(p,n,align)` allocates an object with 'p' pointers, 'n' integers and the given alignment; `alloc(type)` allocates an object of the given type. Variables are declared 'c' style, and are optionally initialized at declaration.

User-defined types are declared as follows:

```
type list {
    int value;
    list next;
}
```

and fields are accessed using java-style "dot" notation, eg

```
list l = alloc(list);
l.value = 0;
l.next = null;
```

At this stage, fields can only be dereferenced to one level, eg `l.next.next` is not valid syntax - you need to introduce a temporary variable to achieve this. Object fields are referenced using syntax like `tmp.int[5]` or `tmp.object[i*3]`, ie like a struct of arrays of the appropriate types.

11.6.2 Syntax

```
script ::= (method|type) ...

method ::= ident "(" { type ident { "," type ident } ...
            ")"
            ( "{" statement ... "}"
              | "intrinsic" "class" name "method" name "signature" "(" java-class {, java class} ")"
            )

type ::= "type" ident "{" field ... "}"
field ::= type ident ";"

statement ::=
    "if" "(" expr ")" block { "elif" "(" expr ")"
        block } [ "else" block ]
    | "while" "(" expr ")" block
    | [ [ type ] ident "=" ] "alloc" "(" expr ","
        expr [ "," expr ] ")" ";"
    | [ ident "=" ] "hash" "(" expr ")" ";"
    | "gc" "(" ")"
    | "spawn" "(" ident [ "," expr ] ... ")" ";"
    | type ident [ "=" expr ] ";"
    | lvalue "=" expr ";"

lvalue ::= ident "=" expr ";"
            | ident "." type "[" expr "]"
```



```

type ::= "int" | "object" | ident

expr ::= expr binop expr
      | unop expr
      | "(" expr ")"
      | ident
      | ident "." type "[" expr "]"
      | ident "." ident
      | int - const
      | intrinsic

intrinsic ::= "alloc" ( "(" expr "," expr "[" expr "]" ")"
                    | type
                    )
            | "(" expr ")"
            | "gc" " "(" ")"

binop ::= "+" | "-" | "*" | "/" | "%" | "&&" | "||" |
        "==" | "!="

unop  ::= "!" | "-"

```

11.7 MMTk Unit Tests

There is a small set of unit tests available for MMTk, using the harness as scaffolding. These tests can be run in the standard test infrastructure using the 'mmtk-unit-tests' test set, or the shell script 'bin/unit-test-mmtk'. Possibly more usefully, they can be run from Eclipse.

To run the unit tests in Eclipse, build the mmtk harness project (see above), and add the directory `testing/tests/mmtk/src` to your build path (navigate to the directory in the package explorer pane in eclipse, right-click → build-path → Use as Source Folder). Either open one of the test classes, or highlight it in the package explorer and press the 'run' button.

Part II

Architecture

This section describes the architecture of Jikes RVM. Jikes RVM can be divided into the following components:

- Core Runtime Services: (thread scheduler, class loader, library support, verifier, etc.) This element is responsible for managing all the underlying data structures required to execute applications and interfacing with libraries.
- Magic: The mechanisms used by Jikes RVM to support low-level systems programming in Java.
- 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.

Chapter 12

Adaptive Optimization System

A comprehensive discussion of the design and implementation of the original Jikes RVM adaptive optimization system is given in the OOPSLA 2000 paper by Arnold, Fink, Grove, Hind and Sweeney. A number of aspects of the system have been changed since 2000, so a better resource is a technical report Nov. 2004 technical report that describes the architecture and implementation in some detail. This section of the userguide is based on section 5 of the 2004 technical report.

The implementation of the Jikes RVM adaptive optimization system uses a number of Java threads: several organizer threads in the runtime measurements component, the controller thread, and the compilation thread. The various threads are loosely coupled, communicating with each other through shared queues and/or the other in memory data structures. All queues in the system are blocking priority queues; if a consumer thread performs a dequeue operation when the queue is empty, it suspends until a producer thread performs an enqueue operation.

The adaptive optimization system performs two primary tasks: selective optimization and profile-directed inlining.

Selective Optimization The goal of selective optimization is to identify regions of code in which the application spends significant execution time (often called “hot spots”), determine if overall application performance is likely to be improved by further optimizing one or more hot spots, and if so to invoke the optimizing compiler and install the resulting optimized code in the virtual machine.

In Jikes RVM, the unit of optimization is a method. Thus, to perform selective optimization, first the runtime measurements component must identify candidate methods (“hot methods”) for the controller to consider. To this end, it installs a listener that periodically samples the currently executing method at every taken yieldpoint. When it is time to take a sample, the listener inspects the thread’s call stack and records a single compiled method id into a buffer. If the yieldpoint occurs in the prologue of a method, then the listener additionally records the compiled method id of the current activation’s caller. If

the taken yieldpoint occurs on a loop backedge or method epilogue, then the listener records the compiled method id of the current method.

When the buffer of samples is full, the sampling window ends. The listener then unregisters itself (stops taking samples) and wakes the sleeping Hot Method Organizer. The Hot Method Organizer processes the buffer of compiled method ids by updating the Method Sample Data. This data structure maintains, for every compiled method, the total number of times that it has been sampled. Careful design of this data structure (`MethodCountData.java`) was critical to achieving low profiling overhead. In addition to supporting lookups and updates by compiled method id, it must also efficiently enumerate all methods that have been sampled more times than a (varying) threshold value. After updating the Method Sample Data, the Hot Method Organizer creates an event for each method that has been sampled in this window and adds it to the controller's priority queue, using the sample value as its priority. The event contains the compiled method and the *total* number of times it has been sampled since the beginning of execution. After enqueueing the last event, the Hot Method Organizer re-registers the method listener and then sleeps until the next buffer of samples is ready to be processed.

When the priority queue delivers an event to the controller, the controller dequeues the event and applies the model-driven recompilation policy to determine what action (if any) to take for the indicated method. If the controller decides to recompile the method, it creates a recompilation event that describes the method to be compiled and the optimization plan to use and places it on the recompilation queue. The recompilation queue prioritizes events based on the cost-benefit computation.

When an event is available on the recompilation queue, the recompilation thread removes it and performs the compilation activity specified by the event. It invokes the optimizing compiler at the specified optimization level and installs the resulting compiled method into the VM.

Although the overall structure of selective optimization in Jikes RVM is similar to that originally described in Arnold et al's OOPSLA 2000 paper, we have made several changes and improvements based on further experience with the system. The most significant change is that in the previous system, the method sample organizer attempted to filter the set of methods it presented to the controller. The organizer passed along to the controller only methods considered "hot". The organizer deemed a method "hot" if the percentage of samples attributed to the method exceeded a dynamically adjusted threshold value. Method samples were periodically decayed to give more weight to recent samples. The controller dynamically adjusted this threshold value and the size of the sampling window in an attempt to reduce the overhead of processing the samples.

Later, significant algorithmic improvements in key data structures and additional performance tuning of the listeners, organizers, and controller reduced AOS overhead by two orders of magnitude. These overhead reductions obviate the need to filter events passed to the controller. This resulted in a more effective system with fewer parameters to tune and a sounder theoretical basis. In general, as we gained experience with the adaptive system implementation, we strove to reduce the number of tuning parameters. We believe that the closer the implementation matches the basic theoretical cost-benefit model, the more likely it will perform well and make reasonable and understandable decisions.

Profile-Directed Inlining Profile-directed inlining attempts to identify frequently traversed call graph edges, which represent caller-callee relationships, and determine whether it is beneficial to recompile the caller methods to allow inlining of the callee methods. In Jikes RVM, profile-directed inlining augments a number of static inlining heuristics. The role of profile-directed inlining is to identify high cost-high benefit inlining opportunities that evade the static heuristics and to predict the likely target(s) of `invokevirtual` and `invokeinterface` calls that could not be statically bound at compile time.

To accomplish this goal, the system takes a statistical sample of the method calls in the running application and maintains an approximation of the dynamic call graph based on this data. The system installs a listener that samples call edges whenever a yieldpoint is taken in the prologue or epilogue of a method. To sample the call edge, it records the compiled method id of the caller and callee methods and the offset of the call instruction in the caller's machine code into a buffer. When the buffer of samples is full, the sampling window ends. The listener then unregisters itself (stops taking samples) and wakes an organizer to update the dynamic call graph with the new profile data. The optimizing compiler's Inline Oracle uses the dynamic call graph to guide its inline decisions. The system currently used is based on Arnold & Grove's CGO 2005 paper. More details of the sampling scheme and the inlining oracle can be found there, or in the source code.

12.1 AOS Controller

A primary design goal for the adaptive optimization system is to enable research in online feedback-directed optimization. Therefore, we require the controller implementation to be flexible and extensible. As we gained experience with the system, the controller component went through several major redesigns to better support our goals.

The controller is a single Java thread that runs an infinite event loop. After initializing AOS, the controller enters the event loop and attempts to dequeue an event. If no event is available, the dequeue operation blocks (suspending the controller thread) until an event is available. All controller events implement an interface with a single method: `process`. Thus, after successfully dequeuing an event the controller thread simply invokes its `process` method and then, the work for that event having been completed, returns to the top of the event loop and attempts to dequeue another event. This design makes it easy to add new kinds of events to the system (and thus, extend the controller's behavior), as all of the logic to process an event is defined by the event's `process` method, not in the code of the controller thread.

A further level of abstraction is accomplished by representing the recompilation strategy as an abstract class with several subclasses. The `process` method of a hot method event invokes methods of the recompilation strategy to determine whether or not a method should be recompiled, and if so at what optimization level. The cost-benefit model itself is also reified in a class hierarchy of models to enable extension and variation. This set of abstractions enable a single controller implementation to execute a variety of strategies.

Another useful mechanism for experimentation is the ability to easily change the input parameters to AOS that define the expected compilation rates and

execution speed of compiled code for the various compilers. By varying these parameters, one can easily cause the default multi-level cost-benefit model to simulate a single-level model (by defining all but one optimization level to be unprofitable). One can also explore other aspects of the system, for example the sensitivity of the model to the accuracy of these parameters. We found this capability to be so useful that the system supports a command line argument (`-X:aos:dna=<filename>`) that causes it to optionally read these parameters from a file.

12.2 Cost Benefit Model

The Jikes RVM Adaptive Optimization System attempts to evaluate the break-even point for each action using an online competitive algorithm. It relies on an analytic model to estimate the costs and benefits of each selective recompilation action, and evaluates the best actions according to the model predictions online. A key advantage of this approach is that it allows a designer to extend the simple "break-even" cost-benefit model to account for more sophisticated adaptive policies, such as selective compilation with multiple optimization levels, on-stack-replacement, and long-running analyses.

In general, each potential action will incur some cost and may confer some benefit. For example, recompiling a method will certainly consume some CPU cycles, but could speed up the program execution by generating better code. In this discussion we focus on costs and benefits defined in terms of time (CPU cycles). However, in general, the controller could consider other measures of cost and benefit, such as memory footprint, garbage allocated, or locality disrupted. The controller will take some action when it estimates the benefit to exceed the cost. More precisely, when the controller wakes at time t , it considers a set of n available actions, the set $A = \{A_1, A_2, \dots, A_n\}$. For any subset S in $P(A)$, the controller can estimate the cost $C(S)$ and benefit $B(S)$ of performing all actions A_i in S . The controller will attempt to choose the subset S that maximizes $B(S) - C(S)$. Obviously $S = \{\}$ has $B(S) = C(S) = 0$; the controller takes no action if it cannot find a profitable course.

In practice, the precise cost and benefit of each action cannot be known; so, the controller must rely on estimates to make decisions.

The basic model the controller uses to decide which method to recompile, at which optimization level, and at what time is as follows.

Suppose that when the controller wakes at time t , and each method m is currently optimized at optimization level $m_i, 0 \leq i \leq k$. Let M be the set of loaded methods in the program. Let A_{jm} be the action "recompile method m at optimization level j , or do nothing if $j = i$."

The controller must choose an action for each m in M . The set of available actions is $Actions = \{A_{jm} | 0 \leq j \leq k, m \in M\}$.

Each action has an estimated cost and benefit: $C(A_{jm})$, the cost of taking action A_{jm} , for $0 \leq j \leq k$ and $T(A_{jm})$, the expected time the program will spend executing method m in the future, if the controller takes action A_{jm} .

For S in $Actions$, define $C(S) = \sum_{s \in S} C(s)$. Given S , for each m in M , define A_{min_m} to be the action A_{jm} in S that minimizes $T(A_{jm})$. Then define $T(S) = \sum_{m \in M} T(A_{min_m})$.

Using these estimated values, the controller chooses the set S that minimizes $C(S) + T(S)$. Intuitively, for each method m , the controller chooses the recompilation level j that minimizes the expected future compilation time and running time of m .

It remains to define the functions C and T for each recompilation action. The basic model models the cost C of compiling a method m at level j as a linear function of the size of m . The linear function is determined by an offline experiment to fit constants to the model.

The basic model estimates that the speedup for any optimization level j is constant. The implementation determines the constant speedup factor for each optimization level offline, and uses the speedup to compute T for each method and optimization level.

We assume that if the program has run for time t , then the program will run for another t units, and then terminate. We further assume program behavior in the future will resemble program behavior in the past. Therefore, for each method we estimate that if no optimization action is performed $T(A_{jm})$ is equal to the time spent executing method m so far.

Let $M = (m_1, \dots, m_k)$ be the k compiled methods. When the controller wakes at time t , each compiled method m has been sampled $\sum m$ times. Let δ be the sampling interval, measured in seconds. The controller estimates that method m has executed $\delta \sum m$ seconds so far, and will execute for another $\delta \sum m$ seconds in the future.

When driving recompilation based on sampling, the controller can limit its attention to the set of methods that were sampled in the previous sampling interval. This optimization does not lose precision; if the number of samples associated with a method has not changed, then the controller's estimate of the method's future execution time will not change. This implies that if the controller were to consider a method that does not appear in the previous sampling interval, the controller would make exactly the same decision it did the last time it considered the method. This optimization, limiting the number of methods the controller must examine in each sampling interval, greatly reduces the amount of work performed by the controller.

Suppose the controller recompiles method m from optimization level i to optimization level j after having seen $\sum m$ samples. Let S_i and S_j be the speedup ratios for optimization levels i and j , respectively. After optimizing at level j , we adjust the sample data to represent the system state as if it had executed method m at optimization level j since program startup. So, we set the new number of samples for m to be $\sum m \cdot (S_i/S_j)$. Thus to compute the time spent in m , we need know only one number, the "effective" number of samples.

12.3 Jikes RVM's compilers

Jikes RVM invokes a compiler for one of three reasons. First, when the executing code reaches an unresolved reference, causing a new class to be loaded, the class loader invokes a compiler to compile the class initializer (if one exists). Second, the system compiles each method the first time it is invoked. In these first two scenarios, the initiating application thread stalls until compilation completes. In the third scenario, the adaptive optimization system can invoke a compiler when profiling data suggests that *recompiling* a method with additional opti-

mizations may be beneficial. The system supports both background and foreground recompilation. With background recompilation (the default), a dedicated thread asynchronously performs all recompilations. With foreground configuration, the system invalidates a compiled method, thus, forcing recompilation at the desired optimization level at the next invocation (stalling the invoking thread until compilation completes).

The system includes two compilers with different tradeoffs between compilation overhead and code quality.

- The goal of the *baseline* compiler is to generate correct code quickly. For example, the IA32 baseline compiler translates bytecodes directly into native code by simulating Java's operand stack. It does not build an intermediate representation and does not perform register allocation, resulting in native code that executes only somewhat faster than bytecode interpretation. However, it does achieve its goal of producing this code quickly, which significantly reduces the initial overhead associated with dynamic compilation.
- The *optimizing* compiler translates bytecodes into an intermediate representation, upon which it performs a variety of optimizations. All optimization levels include linear scan register allocation and BURS-based instruction selection. The compiler's optimizations are grouped into several levels:
 - **Level 0** consists of a set of flow-sensitive optimizations performed on-the-fly during the translation from bytecodes to the intermediate representation and some additional optimizations that are either highly effective or have negligible compilation costs. The compiler performs the following optimizations during IR generation: constant, type, non-null, and copy propagation, constant folding and arithmetic simplification, branch optimizations, field analysis, unreachable code elimination, inlining of trivial methods (A trivial method is one whose body is estimated to take less code space than 2 times the size of a calling sequence and that can be inlined without an explicit guard.), elimination of redundant nullchecks, checkcasts, and array store checks. As these optimizations reduce the size of the generated IR, performing them tends to reduce overall compilation time. Level 0 includes a number of cheap local (The scope of a local optimization is one extended basic block.) optimizations such as local redundancy elimination (common subexpression elimination, loads, and exception checks), copy propagation, constant propagation and folding. Level 0 also includes simple control flow optimizations such as static basic block splitting, peephole branch optimization, and tail recursion elimination. Finally, Level 0 performs simple code reordering, scalar replacement of aggregates and short arrays, and one pass of intraprocedural flow-insensitive copy propagation, constant propagation, and dead assignment elimination.
 - **Level 1** resembles Level 0, but significantly increases the aggressiveness of inlining heuristics. The compiler performs both unguarded inlining of final and static methods and (speculative) guarded inlining of non-final virtual and interface methods. Speculative inlining

is driven both by class hierarchy analysis and online profile data gathered by the adaptive system. In addition, the compiler exploits “preexistence” to safely perform unguarded inlining of some invocations of non-final virtual methods *without* requiring stack frame rewriting on invalidation. It also runs multiple passes of some of the Level 0 optimizations and uses a more sophisticated code reordering algorithm due to Pettis and Hansen.

- **Level 2** augments level 1 with loop optimizations such as normalization and unrolling; scalar SSA-based flow-sensitive optimizations based on dataflow, global value numbering, global common subexpression elimination, redundant and conditional branch elimination; and heap array SSA-based optimizations, such as load/store elimination, and global code placement. **NOTE: many of the O2 optimizations are disabled by default by defining them as O3 optimizations because they are believed to be somewhat buggy.**

The adaptive system uses information about average compilation rate and relative speed of compiled code produced by each compiler/optimization level to make its decisions. These characteristics of the compilers are the key inputs to enable selective optimization to be effective. It allows one to employ a quick executing compiler for infrequently executed methods and an optimizing compiler for the most critical methods. See `org.jikesrvm.adaptive.recompilation.CompilerDNA` for the current values of these input parameters to the adaptive systems cost/benefit model.

12.4 Life Cycle of a Compiled Method

In early implementations of Jikes RVM’s adaptive system, compilation required holding a global lock that serialized compilation and also prevented classloading from occurring concurrently with compilation. This bottleneck was removed in version 2.1.0 by switching to a finer-grained locking discipline to coordinate compilation, speculative optimization, and class loading. Since no published description of this locking protocol exists outside of the source code, we briefly summarize the life cycle of a compiled method here.

When Jikes RVM compiles a method, it creates a compiled method object to represent this particular compilation of the source method. A compiled method has a unique id, and stores the compiled code and associated compiler meta-data. After a brief initialization phase, the compiled method transitions from uncompiled to compiling when compilation begins. During compilation, the optimizing compiler may perform speculative optimizations that can be invalidated by future class loading. Each time the compiler so speculates, it records a relevant entry in an invalidation database. Upon finishing compilation, the system checks to ensure that the current compilation has not already been invalidated by concurrent classloading. If it has not, then the system installs the compiled code, and subsequent invocations will branch to the newly created code.

Each time a class is loaded, the system checks the invalidation database to identify the set of compiled methods to mark as obsolete, because this class-

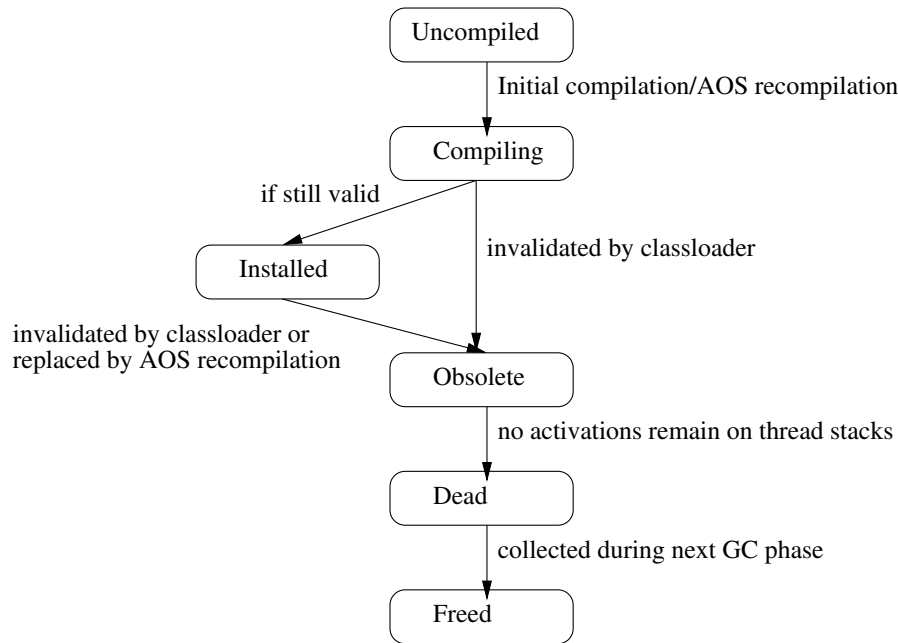


Figure 12.1: life cycle of a compiled method

loading action invalidates speculative optimizations previously applied to that method. A method may transition from either compiling or installed to obsolete due to a classloading-induced invalidation. A method can also transition from installed to obsolete when the adaptive system selects a method for optimizing recompilation and a new compiled method is installed to replace it.

Once a method is marked obsolete, it will never be invoked again. However, before the generated code for the compiled method can be garbage collected, all existing invocations of the compiled method must be complete. A compiled method transitions from obsolete to dead when no invocations of it exist on any thread stack. Jikes RVM detects this as part of the stack scanning phase of garbage collection; as stack frames are scanned, their compiled methods are marked as active. Any obsolete method that is not marked as active when stack scanning completes is marked as dead and the reference to it is removed from the compiled method table. It will then be freed during the next garbage collection.

12.5 Logging and Debugging

Complex non-deterministic systems such as the Jikes RVM adaptive system present challenges for system understanding and debugging. Virtually all of the profiling data collected by the runtime measurements component results from non-deterministic timer-based sampling at taken yieldpoints. The exact timing of these interrupts, and thus, the profile data that drives recompilation decisions, differs somewhat each time an application executes. Furthermore, many of the optimizations in the optimizing compiler rely on online profiles of conditional

branch probabilities, i.e., the probabilities at the point in an execution when the recompilation occurs. Thus, because recompilations can occur at different times during each execution, a method compiled at the same optimization level could be compiled slightly differently on different runs.

The primary mechanism we use to manage this complexity is a record-replay facility for the adaptive system, where online profile data is gathered during one run and used in a subsequent run. More specifically, as methods are dynamically compiled, the system can record this information into a log file. At the end of the run, the system can optionally dump the branch probabilities of all instrumented conditional branches, the profile-derived call graph, and the profile-directed inlining decisions. This log of methods and the files of profile data can then be provided as inputs to a driver program (`org.jikesrvm.tools.opt.OptTestHarness`) that can replay the series of compilation actions, and then optionally execute the program. Usually a fairly rapid binary search of methods being compiled and/or the supporting profile data suffices to narrow the cause of a crash to a small set of actions taken by the optimizing compiler. Although this does not enable a perfectly accurate replay of a previous run, in practice, we have found that it suffices to reproduce almost all crashes caused by bugs in the optimizing compiler.

In addition to this record-replay mechanism, which mainly helps debugging the optimizing compiler, the adaptive system can generate a log file that contains detailed information about the actions of its organizer and controller threads. A sample is shown below:

```
30:...7047728888 Compiled read with baseline compiler in
    0.20 ms
90:...7136817287 Controller notified that read(14402) has
    4.0 samples
92:...7139813016 Doing nothing cost (leaving at baseline)
    to read is 40.0
92:...7139830219 Compiling read cost at O0=40.42, future
    time=49.81
92:...7139842466 Compiling read cost at O1=65.99, future
    time=72.58
92:...7139854029 Compiling read cost at O2=207.44, future
    time=213.49
110:...7166901172 Controller notified that read(14402) has
    9.0 samples
111:...7168378722 Doing nothing cost (leaving at baseline
    ) to read=90.0
111:...7168396493 Compiling read cost at O0=40.42, future
    time=61.54
111:...7168409562 Compiling read cost at O1=65.99, future
    time=80.81
111:...7168421097 Compiling read cost at O2=207.44,
    future time=221.06
111:...7168435937 Scheduling level 0 recompilation of read
    (priority=28.46)
112:...7169879779 Recompiling (at level 0) read
114:...7173293360 Recompiled (at level 0) read
```

```

150:...7227058078 Controller notified that read(14612) has
    5.11 samples
151:...7228691160 Doing nothing cost (leaving at O0) to
    read=51.12
151:...7228705466 Compiling read cost at O1=66.26, future
    time=102.14
151:...7228717124 Compiling read cost at O2=208.29,
    future time=241.24

<....many similar entries....>

998:...8599006259 Controller notified that read(14612) has
    19.11 samples
999:...8599561634 Doing nothing cost (leaving at O0) to
    read=191.13
999:...8599576368 Compiling read cost at O1=54.38, future
    time=188.52
999:...8599587767 Compiling read cost at O2=170.97,
    future time=294.14
999:...8599603986 Scheduling level 1 recompilation of read
    (priority=2.61)
1000:...8601308856 Recompiling (at level 1) read
1002:...8604580406 Recompiled (at level 1) read
1018:...8628022176 Controller notified that read(15312)
    has 18.41 samples
1019:...8629548221 Doing nothing cost (leaving at O1) to
    read=184.14
1019:...8629563130 Compiling read cost at O2=170.97,
    future time=340.06

```

This sample shows an abbreviated subset of the log entries associated with the method `read` of the class `spec.benchmarks._213_javac.ScannerInputStream`, one of the hotter methods of the SPECjvm98 benchmark `_213_javac`. The first pair of numbers are the controller clock (number of timer interrupts since execution began) and the value of the hardware cycle counter (`Time.cycles()`) for the log entry. These log entries show the cost-benefit values computed by the controller for various possible optimization actions and the progression of the method from baseline compilation through two recompilations (level 0 and then at level 1). For example, at time 92, we see four entries that give the estimated total future time (the sum of the compilation cost and the total future execution time in a method) for performing no recompilation and for each optimization level. Because the total future time for not recompiling (40) is less than the other alternatives (49.81, 72.58, and 213.49), the method is not scheduled for recompilation. However, at time 110, the method has been sampled more often. Thus, the total future time estimate is updated, resulting in two recompilation actions (level 0 and level 1) that are more attractive than taking no recompilation action. Because level 0 gives the least future time, this decision is chosen by placing a recompilation event in the recompilation priority queue. The priority for the event is the expected improvement of performing this recompilation, i.e., the difference between the future time for the new level

and the future time for current execution ($90 - 61.54 = 28.46$).

At clock time 150 a similar pattern occurs when considering whether to recompile this method at level 1 or 2; initially recompiling at higher levels is not chosen (clock time 151) until sufficient samples of the method have occurred (clock time 999).

The figure also illustrates how samples of a method at lower optimization level are incorporated into the total samples for a method that has been recompiled. The samples at the lower level are scaled by the relative speed of the two levels as defined by the `CompilerDNA`, and used as the initial number of samples for the higher level. For example, at clock time 100, the baseline compiled version of the method has 9 samples. When the method is recompiled at level 0, these methods are scaled down by 4.26, which is the expected speedup defined by the `CompilerDNA` for going from baseline to level 0, resulting in a value of 2.11. At clock time 160, the level 0 version of method has 5.11 samples, i.e, 3 additional samples of the method have occurred.

12.6 Threading and Yieldpoints

Jikes RVM creates a native thread for each Java thread that is started. Each compiler generates yield points, which are program points where the running thread checks to determine if it should yield to another thread. The compilers insert yield points in method prologues, method epilogues, and on loop backedges.

The adaptive optimization system piggybacks on this yieldpoint mechanism to gather profile data. The thread scheduler provides an extension point by which the runtime measurements component can install listeners that execute each time a yieldpoint is taken. Such listeners primarily serve to sample program execution to identify frequently-executed methods and call edges. Because these samples occur at well-known locations (prologues, epilogues, and loop backedges), the listener can easily attribute each sample to the appropriate Java source method. The Jikes RVM implementation introduces a weakness with this mechanism, in that samples can only occur in regions of code that have yieldpoints. Some low-level Jikes RVM subsystems, such as the thread scheduler and the garbage collector, elide yieldpoints because those regions of code rely on delicate state invariants that preclude thread switching. These uninterruptible regions can distort sampling accuracy by artificially inflating the probability of sampling the first yieldpoint executed after the program leaves an uninterruptible region of code.

Chapter 13

Compilers

- Baseline Compiler
- JNI Compiler: the JNI compiler "compiles" native methods by generating code to transition from Jikes RVM internal calling/register conventions to the native platforms ABI. It is almost completely platform-dependent.
- Optimizing Compiler

13.1 Baseline Compiler

13.1.1 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 boot images to support dynamically loading other compilers).

Roughly two thirds of the baseline compiler is machine-independent. The main file is `BaselineCompiler` and its parent `TemplateCompilerFramework`. The main platform-dependent file is `BaselineCompilerImpl`.

Baseline compilation consists of two main steps: GC map computation (discussed below) and code generation. The code generation in the baseline compilers is mostly straightforward, consisting of a single pass through the bytecodes of the method being compiled.

Differences in the hardware architectures lead to slightly different implementation strategies for the baseline compilers. For example, the IA32 baseline compiler does not try to optimize register usage, instead the bytecode operand stack is held in memory. This leads to bytecodes that push a constant onto the stack, creating a memory write in the generated machine code. The number of memory accesses in the IA32 baseline compiler corresponds directly to the number of bytecodes. In contrast to this, the PPC baseline compiler does some register allocation of local variables (and should probably do even more register allocation to properly exploit the register set).

`TemplateCompilerFramework` contains the main code generation switch statement that invokes the appropriate `emit<bytecode>_` method of `BaselineCompilerImpl`.

13.1.2 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 JSRs; 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.

13.2 Optimizing Compiler

The documentation for the optimizing compiler is organized into the following sections.

- Method Compilation: The fundamental unit for compilation in Jikes RVM is a single method.
- IR: The intermediate representation used by the optimizing compiler.
- BURS: The Bottom-Up Rewrite System (BURS) is used by the optimizing compiler for instruction selection.
- OptTestHarness: A test harness for compilation parameters for specific classes and methods.

13.2.1 Method Compilation

The fundamental unit for optimization in 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 `CompilationPlan` contains all the information necessary to generate machine code for a method. An instance of this class includes, among other fields, the `RVMMethod` to be compiled and the array of `OptimizationPlanElements` which define the compilation steps. The `execute` method of an `CompilationPlan` invokes the optimizing compiler to generate machine code for the method, executing the compiler phases as listed in the plan's `OptimizationPlanElements`.

The `OptimizationPlanner` class defines the standard phases used in a compilation. This class contains a static field, called `masterPlan`, which contains all possible `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 `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 `OptOptions` 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 `OptimizationPlanAtomicElement`. The main work of this class is performed by its phase, an object of type `CompilerPhase`. The `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 `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 `CompilerPhase` can be overridden to return the phase itself rather than a clone of the phase.

13.2.2 IR

The optimizing compiler intermediate representation (IR) is held in an object of type `IR` and includes a list of instructions. Every instruction is classified into one of the pre-defined instruction formats. Each instruction includes an operator and zero or more operands. Instructions are grouped into basic blocks; basic blocks are constrained to having control-flow instructions at their end. Basic blocks fall-through to other basic blocks or contain branch instructions that have a destination basic block label. The graph of basic blocks is held in the `cfg` (control-flow graph) field of `IR`.

This section documents basic information about the intermediate representation. For a tutorial based introduction to the material it is highly recommended that you read the presentation *Jikes RVM Optimizing Compiler Intermediate Code Representation*.

IR Operators

The IR operators are defined by the class `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 `$RVM_ROOT/rvm/src-generated/opt-ir` and defines machine-independent operators. The other resides in `$RVM_ROOT/rvm/src-generated/opt-ir/${arch}` and defines machine-dependent operators, where `${arch}` is the specific instruction architecture of interest.

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.
- **TRAITS**: a set of characteristics of the operator, composed with a bit-wise or (`()`) operator. See `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.

Instruction Format

Every IR instruction fits one of the pre-defined *Instruction Formats*. The Java package `org.jikesrvm.compilers.opt.ir` 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 `Operators` interface and the `Move` instruction format:

```
import org.jikesrvm.compilers.opt.ir.*;
class X {
    void foo(Instruction s) {
        if (Move.conforms(s)) {           // if this instruction
            fits the Move format
            RegisterOperand r1 = Move.getResult(s);
            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 JavadocTM reference for a complete description of the API.

Each fixed-length instruction format is defined in the text file `$RVM_ROOT/rvm/src-generated/opt-ir/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
- **SIZES:** the number of operands defined, defined and used, and used
 - 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 `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 RegisterOperand" "U Val 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 `RegisterOperand`. The use is called `Val` and must be of type `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 RegisterOperand" \
"U Address Operand" "U Method MethodOperand" "U Guard
  Operand opt"
"Param 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 `RegisterOperand` named `Result`.
- **GuardResultCarrier**: provides access to an operand of type `RegisterOperand` named `GuardResult`.
- **LocationCarrier**: provides access to an operand of type `LocationOperand` named `Location`.
- **GuardCarrier**: provides access to an operand of type `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.

13.2.3 BURS

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, doi: 10.1145/151640.151642). The instruction selection rules for each architecture are specified in an architecture-specific file located in `$RVM_ROOT/rvm/src-generated/opt-burs/${arch}`, where `${arch}` is the specific instruction architecture of interest. The rules are used in generating a parser, which transforms the IR.

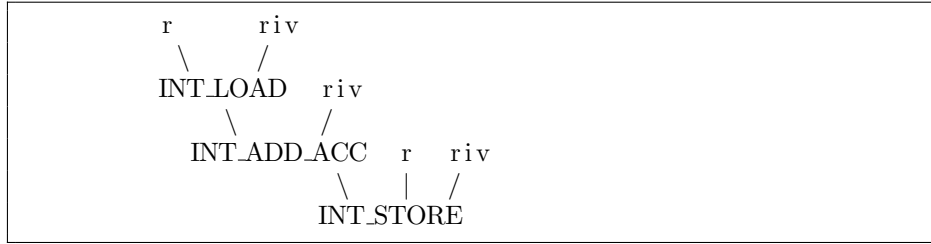
Each rule is defined by a four-line record, consisting of:

- **PRODUCTION**: the tree pattern to be matched. The format of each pattern is explained below.
- **COST**: the cost of matching the pattern as opposed to skipping it. It is a JavaTM expression that evaluates to an integer.
- **FLAGS**: The flags for the operation:
 - **NOFLAGS**: this production performs no operation
 - **EMIT_INSTRUCTION**: this production will emit instructions
 - **LEFT_CHILD_FIRST**: visit child on left-and side of production first
 - **RIGHT_CHILD_FIRST**: visit child on right-hand side of production first
- **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 IA32 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:



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).
- `VL(p)` is used to access the integer constant value associated with the left child of the current (root) node
- `VR(p)` is used to access the integer constant value associated with the right child of the current (root) node
- See `BURS_Common_Helpers` class for definitions of the helper methods

How the above rule basically reads is as follows: 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 `BURS_STATE.java`. Note that if the BURS rules are changed, it is necessary to run `ant real-clean` in order to recreate the auto-generated Java source code for the BURS rules.

For more information on helper functions look at `BURS_Helpers.java`. For more information on the BURS algorithm see `BURS.java`.

-useBootOptions	Use the same OptOptions as the bootimage compiler.
-longcommandline <filename>	Read commands (one per line) from a file
+baseline	Switch default compiler to baseline
-baseline	Switch default compiler to optimizing
-load <class>	Load a class
-class <class>	Load a class and compile all its methods
-method <class><method>[- or <descrip>]	Compile method with default compiler
-methodOpt <class><method>[- or <descrip>]	Compile method with opt compiler
-methodBase <class><method>[- or <descrip>]	Compile method with base compiler
-er <class><method>[- or <descrip>] {args}	Compile with default compiler and execute a method
-performance	Show performance results
-oc	pass an option to the optimizing compiler

Table 13.1: OptTestHarness command line options

Future directions Whilst jburg allows us to do good instruction selection there are a number of areas where it is lacking, e.g. vector operations. We can't write productions for vector operations unless we match an entire tree of operations. For example, it would be nice to write a rule of the form:

$$(r, r) : \text{ADD}(r, r), \text{ADD}(r, r)$$

if say the architecture supported a vector add operation (ie SIMD). Unfortunately we can't have tuples on the LHS of expressions and the comma represents that matching two coverings is necessary. Leupers has shown how to achieve this result with a modified BURS system. Their syntax is:

$$r : \text{ADD}(r, r)$$

$$r : \text{ADD}(r, r)$$

13.2.4 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 prototype-opt image will suit this process using the command line option `-X:aos:initial_compiler=opt` combined with `-X:aos:enable_recompilation=false`. This configuration invokes the optimizing compiler on each method run. The **OptTestHarness** 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.

Examples To use the `OptTestHarness` program:

```
rvm org.jikesrvm.tools.oth.OptTestHarness -class Foo
```

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

```
rvm org.jikesrvm.tools.oth.OptTestHarness -method Foo bar  
-
```

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

```
rvm org.jikesrvm.tools.oth.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 org.jikesrvm.tools.oth.OptTestHarness -oc:O1 -oc:  
    print_final_hir=true -method Foo bar -
```

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

Chapter 14

Core Runtime Services

The Jikes RVM runtime environment implements a variety of services which a Java application relies upon for correct execution. The services include:

- Object Model: The way objects are represented in storage.
- Class and Code Management: The mechanism for loading, and representing classes from class files. The mechanism that triggers compilation and linking of methods and subsequent storage of generated code.
- Thread Management: thread creation, scheduling and synchronization/exclusion
- JNI: Native interface for writing native methods and invoking the virtual machine from native code.
- Calling Conventions: calling conventions used for invoking methods in Jikes RVM
- Exception Management: hardware exception trapping and software exception delivery.
- Bootstrap: getting an initial Java application running in a fully functional Java execution environment

The requirement for many of these runtime services is clearly visible in language primitives such as `new()`, `throw()` and in `java.lang` and `java.io` APIs such as `Thread.run()`, `System.println()`, `File.open()` etc. Unlike conventional Java APIs which merely modify the state of Java objects created by the Java application, implementation of these primitives requires interaction with and modification of the platform (hardware and system software) on which the Java application is being executed.

In addition to the services described above, Jikes RVM also provides some services that are specific to its purpose as a research tool:

- VM Callbacks: Notifications about potentially interesting events in the VM.

14.1 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 `ObjectModel`.

14.1.1 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 following non-functional requirements govern the Jikes RVM object model:

- instance field and array accesses should be as fast as possible,
- null-pointer checks should be performed by the hardware if possible,
- method dispatch and other frequent runtime services 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 resides in a register, compiled code can access the object's fields 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*, resides just before its first element. Thus, compiled code can access array elements via `base + scaled index` addressing.

The Java programming language requires that an attempt to access an object through a null object reference generates a `NullPointerException`. In Jikes RVM, references are machine addresses, and `null` is represented by address 0. On Linux, accesses to both very low and very high memory can be trapped by the hardware, thus all null checks can be made implicit.

14.1.2 Object Header

Logically, every object header contains the following components:

- `textbfTIB` Pointer: The TIB (Type Information Block) holds information that applies to all objects of a type. The structure of the TIB is defined by `TIBLayoutConstants`. 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.
- `textbfHash` Code: Each Java object has an identity hash code. This can be read by `Object.hashCode()` or in the case that this method was overridden, by `System.identityHashCode`. The default hash code is usually

the location in memory of the object, however, with some garbage collectors objects can move. So the hash code remains the same, space in the object header may be used to hold the original hash code value.

- **textbfLock**: Each Java object has an associated lock state. This could be a pointer to a lock object or a direct representation of the lock.
- **textbfArray Length**: Every array object provides a length field that contains the length (number of elements) of the array.
- **textbfGarbage 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.
- **textbfMisc 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 two files:

- **JavaHeader**, which supports TIB access, default hash codes, and locking. It also provides a few bits for use by the memory management subsystem.
- **MiscHeader**, which supports adding additional fields to all objects.

Information that is specific to garbage collection uses the available bits from the Java header. Depending on the chosen garbage collector, the available bits can be accessed via an appropriate class, e.g.:

- **HeaderByte** which provides access to methods for logging and unlogging for various collectors
- **RCHeader** for reference counting garbage collectors
- **ForwardingWord** which provides methods for object forwarding which is used by some copying collectors

14.1.3 Field Layout

Fields tend to be recorded in the Java class file in the order they are declared in the Java source file. We lay out fields in the order they are declared with some exceptions to improve alignment and pack the fields in the object.

Fields of type **double** and **long** benefit from being 8 byte aligned. Every **RVMClass** records the preferred alignment of the object as a whole. We lay out **double** and **long** fields first (and object references if these are 8 bytes long) so that we can avoid making holes in the field layout for alignment. We don't do this for smaller fields as all objects need to be a multiple of 4 bytes in size. When we lay out fields we may create holes to improve alignment. For example, an **int** following a **byte**, we'll create a 3 byte hole following the **byte** to keep the **int** 4 byte aligned. Holes in the field layout can be 1, 2 or 4 bytes in size. As fields are laid out, holes are used to avoid increasing the size of the object. Sub-classes inherit the hole information of their parent, so holes in the parent object can be reused by their children.

14.2 Class and Code Management

The runtime maintains a database of Java instances which identifies the currently loaded class and method base. The classloader class base enables the runtime to identify and dynamically load undefined classes as they are required during execution. All the classes, methods and compiled code arrays required to enable the runtime to operate are pre-installed in the initial boot image. Other runtime classes and application classes are loaded dynamically as they are needed during execution and have their methods compiled lazily. The runtime can also identify the latest compiled code array (and, on occasions, previously generated versions of compiled code) of any given method via this classbase and recompile it dynamically should it wish to do so.

Lazy method compilation postpones compilation of a dynamically loaded class' methods at load-time, enabling partial loading of the class base to occur. Immediate compilation of all methods would require loading of all classes mentioned in the bytecode in order to verify that they were being used correctly. Immediate compilation of these class' methods would require yet more loading and so on until the whole classbase was installed. Lazy compilation delays this recursive class loading process by postponing compilation of a method until it is first called.

Lazy compilation works by generating a stub for each of a class' methods when the class is loaded. If the method is an instance method this stub is installed in the appropriate TIB slot. If the method is static it is placed in a linker table located in the JTOC (linker table slots are allocated for each static method when a class is dynamically loaded). When the stub is invoked it calls the compiler to compile the method for real and then jumps into the relevant code to complete the call. The compiler ensures that the relevant TIB slot/linker table slot is updated with the new compiled code array. It also handles any race conditions caused by concurrent calls to the dummy method code ensuring that only one caller proceeds with the compilation and other callers wait for the resulting compiled code.

14.2.1 Class Loading

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

- **vacant:** The `RVMClass` object for this class has been created and registered and is in the process of being loaded.
- **loaded:** The class's bytecode file has been read and parsed successfully. The modifiers and attributes for the fields have been loaded and the constant pool has been constructed. The class's superclass (if any) and superinterfaces have been loaded as well.
- **resolved:** The superclass and superinterfaces of this class has been resolved. The offsets (whether in the object itself, the JTOC, or the class's TIB) of its fields and methods have been calculated.

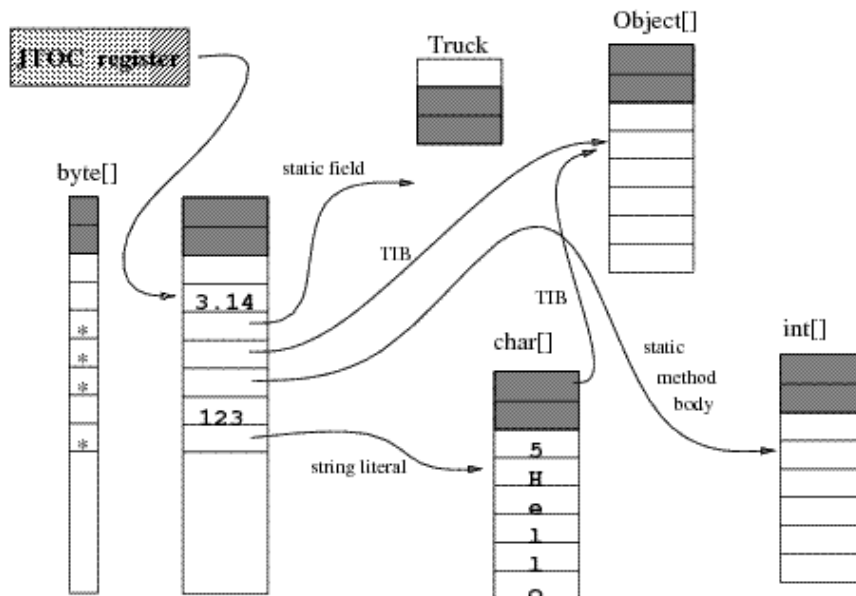
- **instantiated:** The superclass has been instantiated and pointers to the compiled methods or lazy compilation stubs have been inserted into the JTOC (for static methods) and the TIB (for virtual methods).
- **initializing:** The superclass has been initialized and the class initializer is being run.
- **initialized:** The superclass has been initialized and the class initializer has been run.
- **class initializer has failed:** There was an exception during execution of the `<clinit>` method so the class cannot be initialized successfully.

14.2.2 Code Management

A compiled method body is an array of machine instructions (stored as ints on PowerPC™ and bytes on x86-32). The Jikes RVM Table of Contents(JTOC), stores pointers to static fields and methods. However, pointers for instance fields and instance methods are stored in the receiver class's TIB. Consequently, the dispatch mechanism differs between static methods and instance methods.

The JTOC

The JTOC holds pointers to each of Jikes™ RVM's global data structures, as well as literals, numeric constants and references to String constants. 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 the figure below.



Virtual Methods

A TIB contains pointers to the compiled method bodies (executable code) for the virtual methods and other instance 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. A virtual method is dispatched to with the `invokevirtual` bytecode; other instance methods are invoked by the `invokespecial` bytecode.

Static Fields and Methods

Static fields and pointers to static method bodies are stored in the JTOC. Static method dispatch is simpler than virtual dispatch, since a well-known JTOC entry method holds the address of the compiled method body.

Instance Initialization Methods

Pointers to the bodies of instance initialization methods, `<init>`, are stored in the JTOC. (They are always dispatched to with the `invokespecial` bytecode.)

Lazy Method Compilation

Method slots in a TIB or the JTOC may hold either a pointer to the compiled code, or a pointer to the compiled code of the *lazy method invocation stub*. When invoked, the lazy method invocation stub compiles the method, installs a pointer to the compiled code in the appropriate TIB or the JTOC slot, then jumps to the start of the compiled code.

Interface Methods

Regardless of whether or not a virtual method is overridden, virtual method dispatch is still simple since the method will occupy the same TIB offset its defining class and in every sub-class. However, a method invoked through an `invokeinterface` call rather than an `invokevirtual` call, will not occupy the same TIB offset in every class that implements its interface. This complicates dispatch for `invokeinterface`.

The simplest, and least efficient way, of locating an interface method is to search all the virtual method entries in the TIB finding a match. Instead, Jikes RVM uses an *Interface Method Table (IMT)* which resembles a virtual method table for interface methods. 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 `InterfaceInvocation`.

14.3 Thread Management

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

All Java threads (application threads, garbage collector threads, etc.) derive from `RVMThread`. Each `RVMThread` maps directly to one native thread, which may be implemented using whichever C/C++ threading library is in use (currently either pthreads or Harmony threads). Unless `-X:availableProcessors` or `-X:gc:threads` is used, native threads are allowed to be arbitrarily scheduled by the OS using whatever processor resources are available; Jikes™ RVM does not attempt to control the thread-processor mapping at all.

Using native threading gives Jikes™ RVM better compatibility for existing JNI code, as well as improved performance, and greater infrastructure simplicity. Scheduling is offloaded entirely to the operating system; this is both what native code would expect and what maximizes the OS scheduler's ability to optimally schedule Java™ threads. As well, the resulting VM infrastructure is both simpler and more robust, since instead of focusing on scheduling decisions it can take a "hands-off" approach except when Java threads have to be preempted for sampling, on-stack-replacement, garbage collection, `Thread.suspend()`, or locking. The main task of `RVMThread` and other code in `org.jikesrvm.scheduler` is thus to override OS scheduling decisions when the VM demands it.

The remainder of this section is organized as follows. The management of a thread's state is discussed in detail. Mechanisms for blocking and handshaking threads are described. The VM's internal locking mechanism, the Monitor, is described. Finally, the locking implementation is discussed.

14.3.1 Tracking the Thread State

The state of a thread is broken down into two elements:

- Should the thread yield at a safe point?
- Is the thread running Java code right now?

The first mechanism is provided by the `RVMThread.takeYieldpoint` field, which is 0 if the thread should not yield, or non-zero if it should yield at the next safe point. Negative versus positive values indicate the type of safe point to yield at (epilogue/prologue, or any, respectively).

But this alone is insufficient to manage threads, as it relies on all threads being able to reach a safe point in a timely fashion. New Java threads may be started at any time, including at the exact moment that the garbage collector is starting; a starting-but-not-yet-started thread may not reach a safe point if the thread that was starting it is already blocked. Java threads may terminate at any time; terminated threads will never again reach a safe point. Any Java thread may call into arbitrary JNI code, which is outside of the VM's control, and may run for an arbitrary amount of time without reaching a Java safe point. As well, other mechanisms of `RVMThread` may cause a thread to block, thereby making it incapable of reaching a safe point in a timely fashion. However, in each of these cases, the Java thread is "effectively safe" - it is not running Java code that would interfere with the garbage collector, on-stack-replacement, locking, or any other Java runtime mechanism. Thus, a state management system is

needed that would notify these runtime services when a thread is "effectively safe" and does not need to be waited on.

RVMThread provides for the following thread states, which describe to other runtime services the state of a Java thread. These states are designed with extreme care to support the following features:

- Allow Java threads to either execute Java code, which periodically reaches safe points, and native code which is "effectively safe" by virtue of not having access to VM services.
- Allow other threads (either Java threads or VM threads) to asynchronously request a Java thread to block. This overlaps with the `takeYieldpoint` mechanism, but adds the following feature: a thread that is "effectively safe" does not have to block.
- Prevent race conditions on state changes. In particular, if a thread running native code transitions back to running Java code while some other thread expects it to be either "effectively safe" or blocked at a safe point, then it should block. As well, if we are waiting on some Java thread to reach a safe point but it instead escapes into running native code, then we would like to be notified that even though it is not at a safe point, it is now effectively safe, and thus, we do not have to wait for it anymore.

The states used to put these features into effect are listed below.

- **NEW.** This means that the thread has been created but is not started, and hence is not yet running. **NEW** threads are always effectively safe, provided that they do not transition to any of the other states.
- **IN_JAVA.** The thread is running Java code. This almost always corresponds to the OS "runnable" state - i.e. the thread has no reason to be blocked, is on the runnable queue, and if a processor becomes available it will execute, if it is not already executing. **IN_JAVA** thread will periodically reach safe points at which the `takeYieldpoint` field will be tested. Hence, setting this field will ensure that the thread will yield in a timely fashion, unless it transitions into one of the other states in the meantime.
- **IN_NATIVE.** The thread is running either native C code, or internal VM code (which, by virtue of JikesTM RVM's metacircularity, may be written in Java). **IN_NATIVE** threads are "effectively safe" in that they will not do anything that interferes with runtime services, at least until they transition into some other state. The **IN_NATIVE** state is most often used to denote threads that are blocked, for example on a lock.
- **IN_JNI.** The thread has called into JNI code. This is identical to the **IN_NATIVE** state in all ways except one: **IN_JNI** threads have a `JNIEnvironment` that stores more information about the thread's execution state (stack information, etc), while **IN_NATIVE** threads save only the minimum set of information required for the GC to perform stack scanning.
- **IN_JAVA_TO_BLOCK.** This represents a thread that is running Java code, as in **IN_JAVA**, but has been requested to yield. In most cases, when you set `takeYieldpoint` to non-zero, you will also change the state of

the thread from `IN_JAVA` to `IN_JAVA_TO_BLOCK`. If you don't intend on waiting for the thread (for example, in the case of sampling, where you're opportunistically requesting a yield), then this step may be omitted; but in the cases of locking and garbage collection, when a thread is requested to yield using `takeYieldpoint`, its state will also be changed.

- **BLOCKED_IN_NATIVE.** `BLOCKED_IN_NATIVE` is to `IN_NATIVE` as `IN_JAVA_TO_BLOCK` is to `IN_JAVA`. When requesting a thread to yield, we check its state; if it's `IN_NATIVE`, we set it to be `BLOCKED_IN_NATIVE`.
- **BLOCKED_IN_JNI.** Same as `BLOCKED_IN_NATIVE`, but for `IN_JNI`.
- **TERMINATED.** The thread has died. It is "effectively safe", but will never again reach a safe point.

The states are stored in `RVMThread.execStatus`, an integer field that may be rapidly manipulated using compare-and-swap. This field uses a hybrid synchronization protocol, which includes both compare-and-swap and conventional locking (using the thread's Monitor, accessible via the `RVMThread.monitor()` method). The rules are as follows:

- All state changes except for `IN_JAVA` to `IN_NATIVE` or `IN_JNI`, and `IN_NATIVE` or `IN_JNI` back to `IN_JAVA`, must be done while holding the lock.
- Only the thread itself can change its own state without holding the lock.
- The only asynchronous state changes (changes to the state not done by the thread that owns it) that are allowed are `IN_JAVA` to `IN_JAVA_TO_BLOCK`, `IN_NATIVE` to `BLOCKED_IN_NATIVE`, and `IN_JNI` TO `BLOCKED_IN_JNI`.

The typical algorithm for requesting a thread to block looks as follows:

```
thread.monitor().lockNoHandshake();
if (thread is running) {
    thread.takeYieldpoint=1;

    // transitions IN_JAVA -> IN_JAVA_TO_BLOCK, IN_NATIVE
    // ->BLOCKED_IN_NATIVE, etc.
    thread.setBlockedExecStatus();

    if (thread.isInJava()) {
        // Thread will reach safe point soon, or else
        // notify
        // us that it left to native code.
        // In either case, since we are holding the lock,
        // the thread will effectively block on either the
        // safe point
        // or on the attempt to go to native code, since
        // performing
        // either state transition requires acquiring the
        // lock,
        // which we are now holding.
```

```

    } else {
        // Thread is in native code, and thus is "
        //   effectively safe",
        // and cannot go back to running Java code so long
        // as we hold
        // the lock, since that state transition requires
        // acquiring the lock.
    }
}
thread.monitor().unlock();

```

Most of the time, you do not have to write such code, as the cases of blocking threads are already implemented. For examples of how to utilize these mechanisms, see `RVMThread.block()`, `RVMThread.hardHandshakeSuspend()`, and `RVMThread.softHandshake()`. A discussion of how to use these methods follows in the section below.

Finally, the valid state transitions are as follows.

- **NEW to IN_JAVA:** occurs when the thread is actually started. At this point it is safe to expect that the thread will reach a safe point in some bounded amount of time, at which point it will have a complete execution context, and this will be able to have its stack traces by GC.
- **IN_JAVA to IN_JAVA_TO_BLOCK:** occurs when an asynchronous request is made, for example to stop for GC, do a mutator flush, or do an isync on PPC.
- **IN_JAVA to IN_NATIVE:** occurs when the code opts to run in privileged mode, without synchronizing with GC. This state transition is only performed by Monitor, in cases where the thread is about to go idle while waiting for notifications (such as in the case of park, wait, or sleep), and by `org.jikesrvm.runtime.FileSystem`, as an optimization to allow I/O operations to be performed without a full JNI transition.
- **IN_JAVA to IN_JNI:** occurs in response to a JNI downcall, or return from a JNI upcall.
- **IN_JAVA_TO_BLOCK to BLOCKED_IN_NATIVE:** occurs when a thread that had been asked to perform an async activity decides to go to privileged mode instead. This state always corresponds to a notification being sent to other threads, letting them know that this thread is idle. When the thread is idle, any asynchronous requests (such as mutator flushes) can instead be performed on behalf of this thread by other threads, since this thread is guaranteed not to be running any user Java code, and will not be able to return to running Java code without first blocking, and waiting to be unblocked (see **BLOCKED_IN_NATIVE to IN_JAVA** transition).
- **IN_JAVA_TO_BLOCK to BLOCKED_IN_JNI:** occurs when a thread that had been asked to perform an async activity decides to make a JNI downcall, or return from a JNI upcall, instead. In all other regards, this is identical to the **IN_JAVA_TO_BLOCK to BLOCKED_IN_NATIVE** transition.

- **IN_NATIVE to IN_JAVA**: occurs when a thread returns from idling or running privileged code to running Java code.
- **BLOCKED_IN_NATIVE to IN_JAVA**: occurs when a thread that had been asked to perform an async activity while running privileged code or idling decides to go back to running Java code. The actual transition is preceded by the thread first performing any requested actions (such as mutator flushes) and waiting for a notification that it is safe to continue running (for example, the thread may wait until GC is finished).
- **IN_JNI to IN_JAVA**: occurs when a thread returns from a JNI downcall, or makes a JNI upcall.
- **BLOCKED_IN_JNI to IN_JAVA**: same as **BLOCKED_IN_NATIVE to IN_JAVA**, except that this occurs in response to a return from a JNI downcall, or as the thread makes a JNI upcall.
- **IN_JAVA to TERMINATED**: the thread has terminated, and will never reach any more safe points, and thus will not be able to respond to any more requests for async activities.

14.3.2 Blocking and Handshaking

Various VM services, such as the garbage collector and locking, may wish to request a thread to block. In some cases, we want to block all threads except for the thread that makes the request. As well, some VM services may only wish for a "soft handshake", where we wait for each non-collector thread to perform some action exactly once and then continue (in this case, the only thread that blocks is the thread requesting the soft handshake, but all other non-collector threads must "yield" in order to perform the requested action; in most cases that action is non-blocking). A unified facility for performing all of these requests is provided by `RVMThread`.

Four types of thread blocking and handshaking are supported:

- **`RVMThread.block()`**. This is a low-level facility for requesting that a particular thread blocks. It is inherently unsafe to use this facility directly - for example, if thread A calls `B.block()` while thread B calls `A.block()`, the two threads may mutually deadlock.
- **`RVMThread.beginPairHandshake()`**. This implements a safe pair-handshaking mechanism, in which two threads become bound to each other for a short time. The thread requesting the pair handshake waits until the other thread is at a safe point or else is "effectively safe", and prevents it from going back to executing Java code. Note that at this point, neither thread will respond to any other handshake requests until `RVMThread.endPairHandshake()` is called. This is useful for implementing biased locking, but it has general utility anytime one thread needs to manipulate something another thread's execution state.
- **`RVMThread.softHandshake()`**. This implements soft handshakes. In a soft handshake, the requesting thread waits for all non-collector threads to perform some action exactly once, and then returns. If any of those

threads are effectively safe, then the requesting thread performs the action on their behalf. `softHandshake()` is invoked with a `SoftHandshakeVisitor` that determines which threads are to be affected, and what the requested action is. An example of how this is used is found in `org.jikesrvm.mm.mmtk.Collection` and `org.jikesrvm.compilers.opt.runtimesupport.OptCompiledMethod`.

- `RVMThread.hardHandshakeSuspend()`. This stops all threads except for the garbage collector threads and the thread making the request. It returns once all Java threads are stopped. This is used by the garbage collector itself, but may be of utility elsewhere (for example, dynamic software updating). To resume all stopped threads, call `RVMThread.hardHandshakeResume()`. Note that this mechanism is carefully designed so that even after the world is stopped, it is safe to request a garbage collection (in that case, the garbage collector will itself call a variant of `hardHandshakeSuspend()`, but it will only affect the one remaining running Java thread).

14.3.3 The Monitor API

The VM internally uses an OS-based locking implementation, augmented with support for safe lock recursion and awareness of handshakes. The Monitor API provides locking and notification, similar to a Java lock, and may be implemented using either a `pthread_mutex` and a `pthread_cond`, or using Harmony's monitor API.

Acquiring a Monitor lock, or awaiting notification, may cause the calling `RVMThread` to block. This prevents the calling thread from acknowledging handshakes until the blocking call returns. In some cases, this is desirable. For example:

- In the implementation of handshakes, the code already takes special care to use the `RVMThread` state machine to notify other threads that the caller may block. As such, acquiring a lock or waiting for a notification is safe.
- If acquiring a lock that may only be held for a short, guaranteed-bounded length of time, the fact that the thread will ignore handshake requests while blocking is safe - the lock acquisition request will return in bounded time, allowing the thread to acknowledge any pending handshake requests.

But in all other cases, the calling thread must ensure that the handshake mechanism is notified that thread will block. Hence, all blocking `Monitor` methods have both a "NoHandshake" and "WithHandshake" version. Consider the following code:

```
someMonitor.lockNoHandshake();
// perform fast, bounded-time critical section
someMonitor.unlock(); // non-blocking
```

In this code, lock acquisition is done without notifying handshakes. This makes the acquisition faster. In this case, it is safe because the critical section is bounded-time. As well, we require that in this case, any other critical sections protected by `someMonitor` are bounded-time as well. If, on the other hand, the critical section was not bounded-time, we would do:

```

someMonitor.lockWithHandshake();
// perform potentially long critical section
someMonitor.unlock();

```

In this case, the `lockWithHandshake()` operation will transition the calling thread to the `IN_NATIVE` state before acquiring the lock, and then transition it back to `IN_JAVA` once the lock is acquired. This may cause the thread to block, if a handshake is in progress. As an added safety provision, if the `lockWithHandshake()` operation blocks due to a handshake, it will ensure that it does so without holding the `someMonitor` lock.

A special `Monitor` is provided with each thread. This monitor is of the type `NoYieldpointsMonitor` and will also ensure that yieldpoints (safe points) are disabled while the lock is held. This is necessary because any safe point may release the `Monitor` lock by waiting on it, thereby breaking atomicity of the critical section. The `NoYieldpointsMonitor` for any `RVMThread` may be accessed using the `RVMThread.monitor()` method.

Additional information about how to use this API is found in the following section, which discusses the implementation of Java locking.

14.3.4 Thin and Biased Locking

JikesTM RVM uses a hybrid thin/biased locking implementation that is designed for very high performance under any of the following loads:

- Locks only ever acquired by one thread. In this case, biased locking is used, and no atomic operations (like compare-and-swap) need to be used to acquire and release locks.
- Locks acquired by multiple threads but rarely under contention. In this case, thin locking is used; acquiring and releasing the lock involves a fast inlined compare-and-swap operation. It is not as fast as biased locking on most architectures.
- Contended locks. Under sustained contention, the lock is "inflated" - the lock will now consist of data structures used to implement a fast barging FIFO mutex. A barging FIFO mutex allows threads to immediately acquire the lock as soon as it is available, or otherwise enqueue themselves on a FIFO and await its availability.

Thin locking has a relatively simple implementation; roughly 20 bits in the object header are used to represent the current lock state, and compare-and-swap is used to manipulate it. Biased locking and contended locking are more complicated, and are described below.

Biased locking makes the optimistic assumption that only one thread will ever want to acquire the lock. So long as this assumption holds, acquisition of the lock is a simple non-atomic increment/decrement. However, if the assumption is violated (a thread other than the one to which the lock is biased attempts to acquire the lock), a fallback mechanism is used to turn the lock into either a thin or contended lock. This works by using `RVMThread.beginPairHandshake()` to bring both the thread that is requesting the lock and the thread to which the lock is biased to a safe point. No other threads are affected; hence this system

is very scalable. Once the pair handshake begins, the thread requesting the lock changes the lock into either a thin or contended lock, and then ends the pair handshake, allowing the thread to which the lock was biased to resume execution, while the thread requesting the lock may now contend on it using normal thin/contended mechanisms.

Contended locks, or "fat locks", consist of three mechanisms:

1. A spin lock to protect the data structures.
2. A queue of threads blocked on the lock.
3. A mechanism for blocked threads to go to sleep until awoken by being dequeued.

The spin lock is a `org.jikesrvm.scheduler.SpinLock`. The queue is implemented in `org.jikesrvm.scheduler.ThreadQueue`. And the blocking/unblocking mechanism leverages `org.jikesrvm.scheduler.Monitor`; in particular, it uses the `Monitor` that is attached to each thread, accessible via `RVM-Thread.monitor()`. The basic algorithm for lock acquisition is:

```
spinLock.lock();
while (true) {
    if (lock.available()) {
        acquire the lock;
        break;
    } else {
        queue.enqueue(me);
        spinLock.unlock();

        me.monitor().lockNoHandshake();
        while (queue.isQueued(me)) {
            // put this thread to sleep waiting to be
            // dequeued,
            // and do so while the thread is IN_NATIVE to
            // ensure
            // that other threads don't wait on this one for
            // handshakes while we're blocked.
            me.monitor().waitWithHandshake();
        }
        me.monitor().unlock();
        spinLock.lock();
    }
}
spinLock.unlock();
```

The algorithm for unlocking dequeues the thread at the head of the queue (if there is one) and notifies its `Monitor` using the `lockedBroadcastNoHandshake()` method. Note that these algorithms span multiple methods in `org.jikesrvm.scheduler.ThinLock` and `org.jikesrvm.scheduler.Lock`; in particular, `lockHeavy()`, `lockHeavyLocked()`, `unlockHeavy()`, `lock()`, and `unlock()`.

14.4 JNI

14.4.1 Overview

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 code. Jikes RVM implements the 1.1 through 1.4 JNI specifications.
- **Native methods:** This is the mechanism for transitioning from Java code 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 threading:** JNI may be freely used from any Java method. The mechanisms required to make this work are discussed in great detail in Thread Management, and to some extent in the sections that follow.

14.4.2 JNI Functions

All of the 1.1 through 1.4 JNIEnv interface functions are implemented. The functions are defined in the class `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 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 `JNICompiler`.

Calling a JNI function results in the thread attempting to transition from `IN_JNI` to `IN_JAVA` using a compare-and-swap; if this fails, the thread may block to acknowledge a handshake. See Thread Management for more details.

14.4.3 Invoking Native Methods

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

The first mechanism is when RVM calls a method of the class `SysCall`. The native methods thus invoked are defined in one of the C and C++ files of the JikesRVM executable. These native methods are non-blocking system calls or C library services. To implement a syscall, Jikes 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()`). For more details on the mechanics of adding a new syscall to the system, see the header comments of `SysCall.java`. Note again that the syscall methods are NOT JNI methods, but an independent (more efficient) interface that is specific to Jikes RVM.

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 `JNICompiler` to generate a stub routine that manages the transition

between Java bytecode and native code. A JNI call is a GC-safe point, since JNI code cannot freely modify the Java heap.

14.4.4 Interactions with Threading

See the Thread Management subsection for more details on the thread system 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 Jikes RVM, but in the case of JNI, all of the VM's services will know that the thread is "effectively safe" and thus may be ignored for most purposes. Additionally, threads executing JNI code may have handshake actions performed by other threads on their behalf, for example in the case of GC stack scanning. This is not the case with syscalls. As far as Jikes RVM is concerned, a Java thread that enters syscall native code is still executing Java code, but will appear to not reach a safe point until after it emerges from the syscall. This issue may be side-stepped by using the `RVMThread.enterNative()` and `leaveNative` methods, as shown in `org.jikesrvm.runtime.FileSystem`.

14.4.5 Missing Features

- **Native Libraries:** JNI 1.2 requires that the VM specially treat native libraries that contain exported functions named `JNI_OnLoad` and `JNI_OnUnload`. Only `JNI_OnLoad` is currently implemented.
- **JNCompiler:** The only known deficiency in `JNCompiler` is that the prologue and epilogues only handle passing local references to functions that expect a `jobject`; they will not properly handle a `jweak` or a regular global reference. This would be fairly easy to implement.
- **JavaVM interface:** The JavaVM interface has `GetEnv` fully implemented and `AttachCurrentThread` partly implemented, but `DestroyJavaVM`, `DetachCurrentThread`, and `AttachCurrentThreadAsDaemon` are just stubbed out and return error codes. There is no good reason why `AttachCurrentThread` and friends cannot be implemented; it just hasn't been done yet, mostly because there was no easy way to support them prior to the introduction of native threads.
- **Directly-Exported Invocation Interface Functions:** These functions (`GetDefaultJavaVMInitArgs` and `JNI_CreateJavaVM` are partly implemented but `JNI_GetCreatedJavaVMs`) is not implemented. This is because we do not provide a virtual machine library that can be linked against, nor do we support native applications that launch and use an embedded Java VM. There is no inherent reason why this could not be done, but we have not done so yet.

14.4.6 Things JNI Can't Handle

- **atexit routines:** Calling JNI code via a routine run at exit time means calling back into a VM that has been shutdown. This will cause the Jikes RVM to freeze on Intel architectures.

Contributions of any of the missing functionality (and/or associated tests) would be greatly appreciated.

14.5 Exception Management

The runtime has to deal with the relatively small number of hardware signals which can be generated during Java execution. On operating systems other than AIX, an attempt to dereference a `null` value (an access to a null value manifests as a read to a small negative address outside the mapped virtual memory address space) will generate a segmentation fault. This means that the Jikes RVM does not need to generate explicit tests guarding against dereferencing null values and this results in faster code generation for non-exceptioning code.

The Jikes RVM handles the signal and reenters Java so that a suitable Java exception handler can be identified, the stack can be unwound (if necessary) and the handler entered in order to deal with the exception. Failing location of a handler, the associated Java thread must be cleanly terminated.

The Jikes RVM actually employs software traps to generate hardware exceptions in a small number of other cases, for example to trap array bounds exceptions. Once again a software only solution would be feasible. However, since a mechanism is already in place to catch hardware exceptions and restore control to a suitable Java handler the use of software traps is relatively simple to support.

Use of a hardware handler enables the register state at the point of exception to be saved by the hardware exception catching routine. If a Java handler is registered in the call frame which generated the exception this register state can be restored before reentry, avoiding the need for the compiler to save register state around potentially exceptioning instructions. Register state for handlers in frames below the exception frame is automatically saved by the compiler before making a call and so can always be restored to the state at the point of call by the exception delivery code.

The bootloader registers signal handlers which catch `SEGV` and `TRAP` signals. These handlers save the current register state on the stack, create a special handler frame above the saved register state and return into this handler frame executing `RuntimeEntrypoints.deliverHardwareException()`. This method searches the stack from the exceptioning frame (or from the last Java frame if the exception occurs inside native code) looking for a suitable handler and unwinding frames which do not contain one. At each unwind the saved register state is reset to the state associated with the next frame. When a handler is found the delivery code installs the saved register state and returns into the handler frame at the start of the handler block.

The Jikes RVM employs some of the same code used by the hardware exception handler to implement the language primitive `throw()`. This primitive requires a handler to be located and the stack to be unwound so that the handler can be entered. A `throw` operation is always translated into a call to `RuntimeEntrypoints.athrow()` so the unwind can never happen in the handler frame. Hence the register state at the point of re-entry is always saved by the call mechanism and there is no need to generate a hardware exception.

14.6 Bootstrap

Jikes RVM is started up by a boot program written in C, the bootloader. The bootloader is responsible for

- registering signal handlers to deal with the hardware errors generated by Jikes RVM
- establishing the initial virtual memory map employed by Jikes RVM
- mapping the Jikes RVM image files
- installing the addresses of the C wrapper functions which are invoked by the runtime to interact with the underlying operating system into the boot record of at the start of Jikes RVM image area
- setting up the JTOC and TR registers for its RVMThread/ptthread
- switching the pthread into the bootstrap Java stack running the bootstrap Java method in the bootstrap Java thread

At this point all further initialization of Jikes RVM is done either in Java or by employing the wrapper callbacks located in the boot record.

The initial bootstrap routine is `VM.boot()`. It sets up the initial thread environment so that it looks like any other thread created by a call to `Thread.start()` then performs a variety of Java boot operations, including initialising the memory manager subsystem, the runtime compiler, the system classloader and the time classes.

The bootstrap routine needs to rerun class initializers for a variety of the runtime and runtime library classes which are already loaded and compiled into the image file. This is necessary because some of the data generated by these initialization routines will not be valid in the Jikes RVM runtime. The data may be invalid as the host environment that generated the boot image may differ from the current environment.

The boot process enables the Java scheduler and locking system, setting up the data structures necessary to launch additional threads. The scheduler also starts the `FinalizerThread` and multiple garbage collector threads (i.e. multiple instances of `CollectorThread`).

Next, the boot routine boots the JNI subsystem which enables calls to native code to be compiled and executed then re-initialises a few more classes whose init methods require a functional JNI (i.e. `java.io.FileDescriptor`).

Finally, the boot routine loads the boot application class supplied on the rvm command line, creates and schedules a Java main thread to execute this class's main method, then exits, switching execution to the main thread. Execution continues until the application thread and all non-daemon threads have exited. Once there are no runnable threads (other than system threads such as the idle threads, collector threads etc) execution of the RVM runtime terminates and the rvm process exits.

14.6.1 Memory Map

Jikes RVM divides its available virtual memory space into various segments containing either code, or data or a combination of the two. The basic map is as follows:

```

+---> BOOT_IMAGE_START    MAX_MAPPABLE_ADDRESS <---+
|<- SEGMENT_SIZE ->      |
+-----+
+ Platform specific | RVM Image      | RVM Heap      | Plat +
+ ( booter code/ ) | ( initial code )| ( meta data, immortal data )| spec +
+ ( data, shlibs ) | ( & data        )| ( large & small objects  )|    +
+-----+

```

Boot Segment

The bottom segment of the address space is left for the underlying platform to locate the boot program (including statically linked library code) and any dynamically allocated data and library code.

Jikes RVM Image Segment

The next area is the one initialized by the boot program to contain all the initial static data, instance data and compiled method code required in order for the runtime to be able to function. The required memory data is loaded from an image file created by an off line Java program, the boot image writer. This image file is carefully constructed to contain data which, when loaded at the correct address, will populate the runtime data area with a memory image containing:

- a JTOC
- all the TIBs, static method code arrays and static field data directly referenced from the JTOC
- all the dynamic method code arrays indirectly referenced from the TIBS
- all the classloader's internal class and method instances indirectly referenced via the TIBS
- ancillary structures attached to these class and method instances such as class bytecode arrays, compilation records, garbage collection maps etc
- a single bootstrap Java thread instance in which Java execution commences
- a single bootstrap thread stack used by the bootstrap thread.
- a master boot record located at the start of the image load area containing references to all the other key objects in the image (such as the JTOC, the bootstrap thread etc) plus linkage slots in which the booter writes the addresses of its C callback functions.

Jikes RVM Heap Segment

The Jikes RVM heap segment is used to provide storage for code and data created during Java execution. Jikes RVM can be configured to employ various different allocation managers taken from the MMTk memory management toolkit.

14.7 Calling Conventions

14.7.1 Architecture-independent concepts

Stackframe layout and calling conventions may evolve as our understanding of Jikes RVM's performance improves. Where possible, API's should be used to protect code against such changes.

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** - Jikes RVM Table Of Contents. Globally accessible data: constants, static fields and methods.
 - **FP** - Frame Pointer Current stack frame (thread specific).
 - **TR** - Thread register. An object representing the current `RVMThread` instance (the one executing on the CPU containing these registers).
- **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.)

Stack conventions

Stacks grow from high memory to low memory.

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)
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 JavaTM thread must yield the Processor (and yield if threadswitch was requested).

Method epilogue responsibilities

(some of these can be omitted 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.

14.7.2 Architecture-specific calling conventions

The following architecture-specific classes are of interest for calling conventions:

- `StackframeLayoutConstants` for layout of the stack frame
- `JNICompiler` for transition to native code
- `RegisterConstants` for register definitions
- `BaselineConstants` for register usage by the baseline compiler
- `CallingConventions` which expands the calling conventions shortly before register allocation in the optimizing compiler

14.8 VM Callbacks

Jikes™ RVM provides callbacks for many runtime events of interest to the Jikes RVM programmer, such as classloading, VM boot image 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 `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 `Callbacks` by calling the corresponding `add...()` method. For example, to be notified when a class is instantiated, first implement the `Callbacks.ClassInstantiatedMonitor` interface, and then call `Callbacks.addClassInstantiatedMonitor()` with an instance of your class. When any class is instantiated, the `notifyClassInstantiated` method in your instance will be invoked.

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 `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 `Callbacks.addClassInstantiatedMonitor()`).

Since the events for which callbacks are available are internal to the VM, there are 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 `Controller` class in the adaptive system.

14.8.1 An Example: Modifying SPECjvm98 to Report the End of a Run

The SPECjvm®98 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 `Callbacks` interface. The specifics are specified below:

1. Modify `spec/harness/ProgramRunner.java`:

- (a) add an import statement for the `Callbacks` class:

```
import org.jikesrvm.Callbacks;
```

- (b) before the call to `runOnce` add the following:

```
Callbacks.notifyAppRunStart(className, run);
```

- (c) after the call to `runOnce` add the following:

```
Callbacks.notifyAppRunComplete(className, run);
```

2. Recompile the modified file:

```
javac -classpath .:$RVM_BUILD/RVM.classes:$RVM_BUILD/RVM.classes/rvmrt.jar spec/harness/ProgramRunner.java
```

or create a stub version of `Callbacks.java` and place it the appropriate directory structure with your modified file, i.e., `org/jikesrvm/Callbacks.java`

3. Run Jikes RVM as you normally would using the SPECjvm98 benchmarks.

In the current system the `Controller` class will gain control when these callbacks are made and print a message into the AOS log file (by default, placed in Jikes RVM's current working directory and called `AOSLog.txt`).

14.8.2 Another Example: Directing a Recompilation of All Methods During the Application's Execution

Another callback of interest allows an application to direct the VM to recompile all executed methods at a certain point of the application's execution by calling the `recompileAllDynamicallyLoadedMethods` method in the `Callbacks` class. This functionality can be useful to experiment with the performance effects of when compilation occurs. This VM functionality can be disabled using the `DISABLE_RECOMPILE_ALL_METHODS` boolean flag to the adaptive system.

Chapter 15

Magic

Most Java runtimes rely upon the foreign language APIs of the underlying platform operating system to implement runtime behaviour which involves interaction with the underlying platform. Runtimes also occasionally employ small segments of machine code to provide access to platform hardware state. Note that this is expedient rather than mandatory. With a suitably smart Java byte-code compiler it would be quite possible to implement a full Java-in-Java runtime i.e. one comprising only compiled Java code (the JNode project is an attempt to implement a runtime along these lines; the Xerox, MIT, Lambda and TI Explorer Lisp machine implementations and the Xerox Smalltalk implementation were highly successful attempts at fully compiled language runtimes).

This section provides information on ★ magic ★ which is an escape hatch that JikesTM RVM provides to implement functionality that is not possible using the pure JavaTM programming language. For example, the Jikes RVM garbage collectors and runtime system must, on occasion, access memory or perform unsafe casts. The compiler will also translate a call to `Magic.threadSwitch()` into a sequence of machine code that swaps out old thread registers and swaps in new ones, switching execution to the new thread's stack resumed at its saved PC

There are three mechanisms via which the Jikes RVM ★ magic ★ is implemented:

- **Compiler Intrinsics:** Most methods are within class libraries but some functions are built in (that is, intrinsic) to the compiler. These are referred to as intrinsic functions or intrinsics.
- **Compiler Pragmas:** Some intrinsics are do not provide any behaviour but instead provide information to the compiler that modifies optimizations, calling conventions and activation frame layout. We refer to these mechanisms as compiler pragmas.
- **Unboxed Types:** Besides the primitive types, all Java values are boxed types. Conceptually, they are represented by a pointer to a heap object. However, an unboxed type is represented by the value itself. All methods on an unboxed type must be Compiler Intrinsics.

The mechanisms are used to implement the following functionality:

- `RawMemoryAccess`: Unfettered access to memory.
- `Uninterruptible Code`: Declaring code to be uninterruptible.
- `Alternative Calling Conventions`: Declaring different calling conventions and activation frame layout. This is done via annotations, see the `org.vm-magic.pragma` package.

15.1 Compiler Ininsics

A compiler intrinsic will usually generate a specific code sequence. The code sequence will usually be inlined and optimized as part of compilation phase of the optimizing compiler.

Magic

All the methods in `Magic` are compiler intrinsics. 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 JikesTM RVM implementor must be *extremely careful* when writing code that uses `Magic` to circumvent the Java type system. The use of `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 or be guarded by calls to `VM.disableGC` and `VM.enableGC`. The optimizing compiler performs aggressive inlining and code motion, so not explicitly marking such dangerous regions in one of these two manners will lead to disaster.

Since magic is inexpressible in the Java programming language, it is unsurprising that the bodies of `Magic` methods are undefined. Instead, for each of these methods, the Java instructions to generate the code is stored in `GenerateMagic` and `GenerateMachineSpecificMagic` (to generate HIR) and the baseline compilers (to generate assembly code) (Note: 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.). Whenever the compiler encounters a call to one of these magic methods, it inlines appropriate code for the magic method into the caller method.

`sun.misc.Unsafe`

The methods of `sun.misc.Unsafe` are not treated specially by the compilers. The Jikes RVM ships a custom `sun.misc.Unsafe` implementation that implements the operations with Jikes RVM magics and internal helper routines.

15.2 Unboxed Types

If a type is boxed then it means that values of that type are represented by a pointer to a heap object. An unboxed type is represented by the value itself such as `int`, `double`, `float`, `byte` etc.

In the Jikes RVM terminology, an unboxed type is a custom unboxed type. Normal Java primitives such as `int` are never referred to as unboxed types.

The Jikes RVM also defines a number of unboxed types. Due to a limitation of the way the compiler generates code the Jikes RVM must define an unboxed array type for each unboxed type. The unboxed types are:

- `org.vmmagic.unboxed.Address`
- `org.vmmagic.unboxed.Extent`
- `org.vmmagic.unboxed.ObjectReference`
- `org.vmmagic.unboxed.Offset`
- `org.vmmagic.unboxed.Word`
- `org.jikesrvm.ArchitectureSpecific.Code`

Values of unboxed types appear only in the virtual machine's stack, registers, or as fields/elements of class/array instances.

Unboxed types may inherit from `Object` but they are not objects. As such there are some restrictions on the use of unboxed types:

- A unboxed type instance must not be passed where an `Object` is expected. This will type-check, but it is not what you want. A corollary is to avoid overloading a method where the two overloaded versions of the method can only be distinguished by operating on an `Object` versus an unboxed type. The optimizing compiler can detect *some* invalid uses of unboxed types.
- An unboxed type must not be synchronized on.
- They have no virtual methods.
- They do not support lock operations, generating hashcodes or any other method inherited from `Object`.
- All methods must be compiler intrinsics.
- Avoid making an array of an unboxed type. Instead represent it by the array version of unboxed type. i.e. `org.vmmagic.unboxed.Address[]` should be replaced with `org.vmmagic.unboxed.AddressArray` but `org.vmmagic.unboxed.AddressArray[]` is fine.

15.3 Raw Memory Access

The type `org.vmmagic.Address` is used to represent a machine-dependent address type. `org.vmmagic.Address` is an unboxed 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 addresses are more appropriately considered unsigned. The difference is problematic since an unsigned comparison on `int` is inexpressible in the Java programming language.

To overcome these problems, instances of `org.vmmagic.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 RVM 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 either a 32-bit or 64-bit unsigned integer. Since an address is an unboxed type it must obey the rules outlined in Unboxed Types.

15.4 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.

- 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.
- 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.
- 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*
- *multianewarray*
- Uninterruptible code cannot cause class loading and thus must not contain unresolved *getstatic*, *putstatic*, *getfield*, *putfield*, *invokevirtual*, or *invokestatic* bytecodes.
- Uninterruptible code cannot contain calls to interruptible code. As a consequence, it is illegal to override an uninterruptible virtual method with an interruptible method.
- Uninterruptible methods cannot be **synchronized**. If you need synchronization in an uninterruptible method, you must use one of the internal locks or synchronization primitives.

We have augmented the baseline compiler to print a warning message when one of these restrictions is violated. The optimizing compiler currently does not check for uninterruptibility violations. Consequently, it is a good idea to compile a boot image with the baseline compiler (e.g. using *prototype-opt*) after modifying uninterruptible code.

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 (or VM bug) and do not flag bytecodes that might result in one of these exceptions being raised as a violation of uninterruptibility.

In a few cases it is necessary to modify the conditions of checking for uninterruptibility to avoid spurious warning messages. This should be done with extreme care. The checking conditions for a particular method can be modified by using one of the following annotations:

- `org.vmmagic.pragma.UninterruptibleNoWarn` - disables checking for uninterruptibility violations but behaves like `org.vmmagic.pragma.Uninterruptible` otherwise. Used for methods that need to be uninterruptible but are **only** executed when writing the boot image.
- `org.vmmagic.pragma.Unpreemptible` - instructs the JVM to avoid inserting operations that could trigger garbage collection or thread switching but does not disallow them. Calls to preemptible code will cause warnings. This is used for code that is involved in thread scheduling, locking or the creation of exception objects.
- `org.vmmagic.pragma.UnpreemptibleNoWarn` - used for unpreemptible code that calls interruptible code.

Do not use the annotation `org.vmmagic.pragma.LogicallyUninterruptible`. Its usage is being phased out.

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

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

- All object constructors are interruptible, since they can only be invoked as part of the implementation of the new bytecode.
- If a method is annotated with `org.vmmagic.pragma.Interruptible` then it is interruptible.
- If none of the above rules apply and a method is annotated with `org.vmmagic.pragma.Uninterruptible`, then it is uninterruptible.
- If none of the above rules apply and the declaring class is annotated with `org.vmmagic.pragma.Uninterruptible` then it is uninterruptible.

Whether to annotate a class or a method with `org.vmmagic.pragma.Uninterruptible` 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 annotating the class is preferred.

Chapter 16

MMTk

The garbage collectors for Jikes RVM are provided by MMTk. The document MMTk: The Memory Manager Toolkit describes MMTk and gives a tutorial on how to use and edit it and is the best place to start. An updated version of the tutorial is available in this guide. A detailed description of the call chain from the compilers through to MMTk here is another good place to start understanding how MMTk integrates with Jikes RVM. Anatomy of a Garbage Collector describes the major building blocks of an MMTk collector and Scanning Objects in Jikes RVM describes how objects are scanned for their pointer fields during GC. MMTk also has a pure Java test harness that allows development of garbage collectors in an IDE like eclipse.

Jikes RVM can be configured to employ various different allocation managers taken from the MMTk memory management toolkit. Managers divide the available space up as they see fit. However, they normally subdivide the available address range to provide:

- a metadata area which enables the manager to track the status of allocated and unallocated storage in the rest of the heap.
- an immortal data area used to service allocations of objects which are expected to persist across the whole lifetime of the Jikes RVM runtime (e.g. the boot image)
- a large object space used to service allocations of objects which are larger than some specified size (e.g. a virtual memory page) - the large object space may employ a different allocation and reclamation strategy to that used for other objects.
- a small object allocation area which may be divided into e.g. two semi spaces, a nursery space and a mature space, a set of generations, a non-relocatable buddy hierarchy etc depending upon the allocation and reclamation strategy employed by the memory manager.
- separate spaces for code. These are designed to exclude performance problems that can occur on some architectures when code and data are mixed. See this paper for the original motivation and experiments for a separate code space.

Virtual memory pages are lazily mapped into Jikes RVM's memory image as they are needed.

The main class which is used to interface to the memory manager is called **Plan**. Each flavor of the manager is implemented by substituting a different implementation of this class. Most plans inherit from class **StopTheWorldGC** which ensures that all active mutator threads (i.e. ones which do not perform the job of reclaiming storage) are suspended before reclamation is commenced. The argument passed to `-X:gc:threads` determines the number of parallel collector threads that will be used for collection.

Generational collectors employ a plan which inherits from class **Generational**. Inter alia, this class ensures that a write barrier is employed so that updates from old to new spaces are detected.

Jikes RVM may also use the GCSPy visualization framework. GCSPy allows developers to observe the behavior of the heap and related data structures.

16.1 Anatomy of a Garbage Collector

**** Work in progress, contributions appreciated ****

This page gives a brief outline of the major control flows in the execution of a garbage collector in MMTk. For simplicity, we focus on the MarkSweep collector, although much of the discussion will be relevant to other collectors.

This page assumes you have a basic knowledge of garbage collection. For those that don't, please see one of the standard texts such as The Garbage Collection Handbook.

16.1.1 Structure of a Plan

An MMTk Plan is required to provide 5 classes. They are required to have consistent names which start with the same name and have a suffix that indicates which class it inherits from. in the case of the MarkSweep plan, the name is "MS".

- **MS** - this is a singleton class that is a subclass of `org.mmtk.plan.Plan`. This class encapsulates data structures that are shared among multiple threads.
- **MSMutator** - subclass of `org.mmtk.plan.MutatorContext`. This class encapsulates data structures that are local to a single mutator thread. In the case of Jikes RVM, a Thread is actually a subclass of this class for efficiency reasons.
- **MSCollector** - subclass of `org.mmtk.plan.CollectorContext`. This provides thread-local data structures specific to a garbage collector thread.
- **MSConstraints** - subclass of `org.mmtk.plan.PlanConstraints`. This provides configuration information that the host virtual machine might need. It is separated out from the Plan class in order to prevent circular class loading dependencies.

- **MSTraceLocal** - subclass of `org.mmtk.plan.TraceLocal`. This provides thread-local data structures specific to a particular way of traversing the heap. In a simple collector like MarkSweep, there is only one of these classes, but in more complex collectors there may be several. For example, in a generational collector, there will be one **TraceLocal** class for a nursery collection, and another for a full-heap collection.

The basic architecture of MMTk is that virtual address space is divided into chunks (of 4MB in a 32-bit memory model) that are managed according to a specific *policy*. A policy is implemented by an instance of the **Space** class, and it is in the policy class that the mechanics of a particular mechanism (like mark-sweep) is implemented. The task of a **Plan** is to create the policy (**Space**) objects that manage the heap, and to integrate them into the MMTk framework. MMTk exposes some of this memory management policy to the host VM, by allowing the VM to specify an allocator (represented by a small integer) when allocating space. The interface exposed to the VM allows it to choose whether an object will move during collection or not, whether the object is large enough to require special handling etc. The MMTk plan is free (within the semantic guarantees exposed to the VM) to direct each of these allocators to a particular policy.

16.1.2 Policies

A policy describes how a range of virtual address space is managed. The base class of all policies is `org.mmtk.policy.Space`, and a particular instance of a policy is known generically as a space. The static initializer of a **Plan** and its subclasses define the spaces that make up an MMTk plan.

MS.java

```
public static final MarkSweepSpace msSpace = new
    MarkSweepSpace("ms", VMRequest.discontiguous());
public static final int MARKSWEEP = msSpace.
    getDescriptor();
```

In this code fragment, we see the **MS** plan defined. Note that we generally also define a **static final** space descriptor. This is an optimization that allows some rapid operations on spaces.

A **Space** is a global object, shared among multiple mutator threads. Each policy will also have one or more thread-local classes which provide unsynchronized allocation. These classes are subclasses of `org.mmtk.utility.alloc.Allocator`, and in the case of MarkSweep, it is called **MarkSweepLocal**. Instances of **MarkSweepLocal** are created as part of a mutator context, like this

MSMutator.java

```
protected MarkSweepLocal ms = new MarkSweepLocal(MS.
    msSpace);
```

The design pattern is that the local Allocator will allocate space from a thread-local buffer, and when that is exhausted it will allocate a new buffer from the global **Space**, performing appropriate locking. The constructor of the **MarkSweepLocal** specifies the space from which the allocator will allocate global memory.

16.1.3 Allocation

MMTk provides two methods for allocating an object. These are provided by the `MSMutator` class, to give each plan the opportunity to use fast, unsynchronized thread-local allocation before falling back to a slower synchronized slow-path. The version implemented in `MarkSweep` looks like this:

`MSMutator.java`

```
public Address alloc(int bytes, int align, int offset,
    int allocator, int site) {
    if (allocator == MS.ALLOC_DEFAULT) {
        return ms.alloc(bytes, align, offset);
    }
    return super.alloc(bytes, align, offset, allocator,
        site);
}
```

The basic structure of this method is common to all MMTk plans. First they decide whether the operation applies to this level of abstraction (`if (allocator == MS.ALLOC_DEFAULT)`), and if so, delegate to the appropriate place, otherwise pass it up the chain to the super-class. In the case of `MarkSweep`, `MSMutator` delegates the allocation to its thread-local `MarkSweepLocal` object `ms`. The `alloc` method of `MarkSweepLocal` is inherited from `SegregatedFreeListLocal` (mark-sweep is not the only way of managing free-list allocation), and looks like this

`SegregatedFreeListLocal.java (simplified)`

```
public final Address alloc(int bytes, int align, int
    offset) {
    int sizeClass = getSizeClass(bytes);
    Address cell = freeList.get(sizeClass);
    if (!cell.isZero()) {
        freeList.set(sizeClass, cell.loadAddress());
        /* Clear the free list link */
        cell.store(Address.zero());
        return cell;
    }
    return allocSlow(bytes, align, offset);
}
```

This is a standard pattern for thread-local allocation: first we look in the thread-local space (line 3), and if successful return the result (lines 4-8). If unsuccessful, we request space from the global policy via the method `Allocator.allocSlow`. This is the common interface that all Allocators use to request space from the global policy. This will eventually call the allocator-specific `allocSlowOnce` method. The workings of the `allocSlowOnce` method are very policy-specific, so not appropriate to look at at this stage, but eventually all policies will attempt to acquire fresh virtual memory via the `Space.acquire` method. `Space.acquire` is the only correct way for a policy to allocate new virtual memory for its own use.

Space.java (simplified)

```

public final Address acquire(int pages) {
    pr.reservePages(pages);
    // Poll, either fixing budget or requiring GC
    if (VM.activePlan.global().poll(false, this)) {
        VM.collection.blockForGC();
        return Address.zero(); // GC required, return failure
    }
    // Page budget is ok, try to acquire virtual memory
    Address rtn = pr.getNewPages(pagesReserved, pages,
        zeroed);
    if (rtn.isZero()) { // Failed, so force a GC
        boolean gcPerformed = VM.activePlan.global().poll(
            true, this);
        VM.collection.blockForGC();
        return Address.zero();
    }
    return rtn;
}

```

The logic of `space.acquire` is:

- First, poll the plan to find out whether the heap is full. This logic is performed by the plan, because it has knowledge of copy reserves etc.
- The `poll` method will request a GC if required, and return `true` if it has done so.
- Then we wait for GC if required. `poll` can't wait, because it is called in circumstances that aren't GC safe.
- If `Plan.poll(...)` returns `false` (we are within the allowed heap size), we call `pr.getNewPages` to allocate virtual memory. At this stage we can find that we have run out of virtual memory, and if so, we force a GC
- If a GC is performed, we return `Address.zero()`, rather than retrying locally. In many plans, the next allocation request will be satisfied by re-using space in a page that already belongs to a policy, so the post-GC allocation must be performed further up in the call stack. The retry logic is handled in `Allocator.allocSlowInline`.

Allocator.java (simplified)

```

public final Address allocSlowInline(int bytes, int
    alignment, int offset) {
    boolean emergencyCollection = false;
    while (true) {
        Address result = allocSlowOnce(bytes, alignment,
            offset);
        if (!result.isZero()) {
            return result;
        }
    }
}

```

```

    if (emergencyCollection) {
        VM.collection.outOfMemory();
    }
    emergencyCollection = Plan.isEmergencyCollection();
}

```

This code fragment shows the retry logic in the allocator. We try allocating using `allocSlowOnce`, which may recycle partially-used blocks and eventually call `Space.acquire`. If a GC occurred, we try again. Eventually the plan will request an emergency collection which will (for example) cause soft references to be dropped. If this fails we throw an `OutOfMemoryError`.

16.1.4 Collection

Scheduling

In a stop-the-world garbage collector like MarkSweep, the mutator threads run until memory is exhausted, then all mutator threads are suspended, the collector threads are activated, and they perform a garbage collection. After the GC is complete, the collector threads are suspended and the mutator threads resume. MMTk also has some support for concurrent collectors, in which one or more collector threads can be scheduled to run alongside the mutator, either exclusively or in addition to (hopefully briefer) stop-the-world phases.

Thread scheduling in MMTk is handled by a GC controller thread, implemented in the singleton class `org.mmtk.plan.ControllerCollectorContext` held in the static field `Plan.controlCollectorContext`. Whenever a collection is initiated, it is done by calling methods on this object.

Initiating

As mentioned above, every attempt to allocate fresh virtual memory calls the current plan's `poll(...)` method. This initiates a GC by calling `controlCollectorContext.request()`, which in a stop-the-world collector like MarkSweep pauses the mutator threads and then wakes the collector threads. The main loop of the garbage collector is simply the `run()` method of `ParallelCollector`, shown below.

ParallelCollector

```

public void run() {
    while(true) {
        park();
        collect();
    }
}

```

The `collect()` method is specific to the type of collector, and in `StopTheWorldCollector` it looks like this

StopTheWorldCollector

```

public void collect() {

```

```

    Phase.beginNewPhaseStack(Phase.scheduleComplex(global()
        .collection));
}

```

Collector Phases

Every garbage collection consists of a series of steps. Each step is either executed once (e.g. updating the mark state before marking the heap), or in parallel on all available collector threads (e.g. the parallel mark phase). The actual work of a step is done by the `collectionPhase` method of the global, collector or mutator class of a plan.

In early versions of MMTk, the main collection method was a template method, calling individual methods for each phase of the collection. As the number of collectors in MMTk grew, this became unwieldy and has been replaced with a configurable mechanism of phases.

The class `org.mmtk.plan.Simple` defines the basic structure of most of MMTk's garbage collectors. First it defines the phases themselves,

Simple.java

```

public static final short SET_COLLECTION_KIND = Phase.
    createSimple("set - collection - kind", null);
public static final short INITIATE           = Phase.
    createSimple("initiate", null);
public static final short PREPARE           = Phase.
    createSimple("prepare");
...

```

Each phase of the collection is represented by a 16-bit integer, an index into a table of `Phase` objects. Simple phases are scheduled, and combined into sequences, or complex phases.

Simple.java

```

/** Ensure stacks are ready to be scanned */
protected static final short prepareStacks = Phase.
    createComplex("prepare - stacks", null,
        Phase.scheduleMutator    (PREPARE_STACKS),
        Phase.scheduleGlobal    (PREPARE_STACKS));

```

A simple phase can be scheduled in one of 4 ways:

- Global. One collector thread is chosen to run the `collectionPhase` method of the global `Plan` object.
- Collector. All collector threads run `collectionPhase` of the plan's `CollectorContext` object(s).
- Mutator. The collector threads run in parallel and iterate over the available `MutatorContext` objects (ie the mutator threads), and run the mutator's `collectionPhase` method. Note that the collector threads are performing work on a per-mutator basis, because in general the mutator threads are stopped at this point.

- Concurrent. The controller is requested to start a concurrent collector thread.

Between every phase of a collection, the collector threads rendezvous at a synchronization barrier. The actual execution of a collector's phases is done in the method `Phase.processPhaseStack`. This method handles resuming a concurrent collection as well as running a full stop-the-world collection.

The actual work of a collection phase is done (as mentioned above) in the `collectionPhase` method of the major `Plan` classes.

MS.java

```
@Inline
@Override
public void collectionPhase(short phaseId) {
    if (phaseId == PREPARE) {
        super.collectionPhase(phaseId);
        msTrace.prepare();
        msSpace.prepare(true);
        return;
    }
    if (phaseId == CLOSURE) {
        msTrace.prepare();
        return;
    }
    if (phaseId == RELEASE) {
        msTrace.release();
        msSpace.release();
        super.collectionPhase(phaseId);
        return;
    }
    super.collectionPhase(phaseId);
}
```

This excerpt shows how the global MS plan implements `collectionPhase`, illustrating the key phases of a simple stop-the-world collector. The prepare phase performs tasks such as changing the mark state, the closure phase performs a transitive closure over the heap (the mark phase of a mark-sweep algorithm) and the release phase performs any post-collection steps. Where possible, a plan is structured so that each layer of inheritance deals only with the objects it creates, i.e. the MS class operates on the `msSpace` and delegates work on all other spaces to the super-class where they are defined. By convention the `PREPARE` phase is performed outside-in (super-class preparation first) and `RELEASE` is done inside-out (local first, super-class second).

Tracing the heap

The main operation of a tracing collector is the transitive closure operation where all (or a subset) of the object graph is visited. Some collectors such as generational collectors perform these operations in more than one way, e.g. a nursery collection in a generational collector does not trace through pointers into the mature space, while a full-heap collection does. All MMTk collectors

are designed to run using several parallel threads, using data structures that have unsynchronized thread-local and synchronized global components in the same way as MMTk's policy classes.

MMTk's trace operation uses the following terminology:

- An *edge* is a reference in the heap from one reference field to the object (or node) it points to.
- *Tracing* an object is the policy-defined operation performed by the collector on an object. In a mark-sweep policy this means setting the mark state of the object. In a copying policy this means moving the object to its new location.
- *Scanning* is the process of identifying the reference fields of an object and processing the objects reachable from each of them.

Each distinct transitive closure operation is defined as a subclass of *TraceLocal*. The closure is performed in the *collectionPhase* method of the plan-specific *CollectorContext* class

MSCollector.java

```
public void collectionPhase(short phaseId, boolean
    primary) {
    ...
    if (phaseId == MS.CLOSURE) {
        fullTrace.completeTrace();
        return;
    }
    ...
}
```

The initial starting point for the closure is computed by the `STACK_ROOTS` and `ROOTS` phases, which add root locations to a buffer by calling `TraceLocal.reportDelayedRootEdge`. The closure operation proceeds by invoking `traceObject` on each root location (in method `processRootEdge`), and then invoking `scanObject` on each heap object encountered. Note that the `CLOSURE` operation is performed multiple times in each GC, due to processing of reference types.

16.2 Memory Allocation in Jikes RVM

The way that objects are allocated in Jikes RVM can be difficult to grasp for someone new to the code base. This document provides a detailed look at some of the paths through the JikesRVM - MMTk interface code to help bootstrap understanding of the process. The process and code illustrated below is current as of March 2011, svn revision 16052 (between JikesRVM 3.1.1 and 3.1.2).

16.2.1 Memory Manager Interface

The best starting place to understand the allocation sequence is in the class `org.jikesrvm.mm.mminterface.MemoryManager`, which is a facade class for the MMTk allocators. MMTk provides a variety of memory management plans

which are designed to be independent of the actual language being implemented. The **MemoryManager** class orchestrates the services of MMTk to allocate memory, and adds the structure necessary to make the allocated memory into Java objects.

The method **allocateScalar** is where all scalar (ie non-array) objects are allocated. The parameters of this method specify the object to be allocated in sufficient detail that when this method is compiled by the opt compiler, all of the parameters are compile-time constants, allowing maximum optimization. Working through the body of the method,

```
Selected.Mutator mutator = Selected.Mutator.get();
```

As mentioned above, MMTk provides many different memory management plans, one of which is selected at build time. This call acquires a pointer to the thread-local per-mutator component of MMTk. Much of MMTk's performance comes from providing unsynchronized thread-local data structures for the frequently used operations, so rather than provide a single interface object, it provides a per-thread interface object for both mutator and collector threads.

```
allocator = mutator.checkAllocator(org.jikesrvm.runtime.  
    Memory.alignUp(size, MIN_ALIGNMENT), align, allocator)  
    ;
```

An MMTk plan in general provides several spaces where objects can be allocated, each with their own characteristics. Jikes RVM is free to request allocation in any of these spaces, but sometimes there are constraints only available on a per-allocation basis that might force MMTk to override Jikes RVM's request. For example, Jikes RVM may specify that objects allocated by a particular class are allocated in MMTk's non-moving space. At execution time, one such object may turn out to be too large for allocation in the general non-moving space provided by that particular plan, and so MMTk needs to promote the object to the Large Object Space (LOS), which is also non-moving, but has high space overheads. This call will generally compile down to 0 or a small handful of instructions.

```
Address region = allocateSpace(mutator, size, align,  
    offset, allocator, site);
```

This calls a method of **MemoryManager**, common to all allocation methods (for Arrays and other special objects), that calls

```
Address region = mutator.alloc(bytes, align, offset,  
    allocator, site);
```

to actually allocate memory from the current MMTk plan.

```
Object result = ObjectModel.initializeScalar(region, tib,  
    size);
```

Now we call the Jikes RVM object model to initialize the allocated region as a scalar object, and then

```
mutator.postAlloc(ObjectReference.fromObject(result),  
    ObjectReference.fromObject(tib), size, allocator);
```

we call MMTk's `postAlloc` method to perform initialization that can only be performed after an object has been initialized by the virtual machine.

16.2.2 Compiler integration

The `allocateScalar` method discussed above is only actually called from one place, the method `resolvedNewScalar(int ...)` in the class `org.jikesrvm.runtime.RuntimeEntrypoints`. This class provides methods that are accessed directly by the compilers, via fields in the `org.jikesrvm.runtime.Entrypoints` class. The 'resolved' part of the method name indicates that the class of object being allocated is resolved at compile time (recall that the Java Language Spec requires that classes are only loaded, resolved etc when they are needed - sometimes it's necessary to compile code that performs classloading and then allocate the object).

`RuntimeEntrypoints` also contains an overload, `resolvedNewScalar(RVMClass)`, that is used by the reflection API to allocate objects. It's instructive to look at this method, as it performs essentially the same operations as the compiler when compiling the call to `resolvedNewScalar(int ...)`.

Working backwards from this point requires delving into the individual compilers.

Baseline Compiler

There is a different baseline compiler for each architecture. The relevant code in the baseline compiler for the ia32 architecture is in the class `org.jikesrvm.compilers.baseline.ia32.BaselineCompilerImpl`. The method `e-mit.resolved_new(RVMClass)` is responsible for generating code to execute the *new* bytecode when the target class is already resolved. Looking at this method, you can see it does essentially what the `resolvedNewScalar(RVMClass)` method in `RuntimeEntrypoints` does, then generates machine code to perform the call to the `resolvedNewScalar` entrypoint. Note how the work of calculating the size, alignment etc of the object is performed by the compiler, at compile time. Similar code exists in the PPC baseline compiler.

Optimizing Compiler

The optimizing compiler is paradoxically somewhat simpler than the baseline compiler, in that injection of the call to the entrypoint is done in an architecture independent level of compiler IR. (An overview of the Jikes RVM optimizing compiler can be found in the paper *The Jalapeño Dynamic Optimizing Compiler for Java*).

In HIR (the high-level Intermediate Representation), allocation is expressed as a 'new' opcode. During the translation from HIR to LIR (Low-level IR), this and other opcodes are translated into instructions by the class `org.jikesrvm.compilers.opt.hir2lir.ExpandRuntimeServices`. The method `perform(IR)` performs this translation, selecting particular operations via a large switch statement. The `NEW_opcode` case performs the task we're interested in, doing essentially the same job as the baseline compiler, but generating IR rather than machine instructions. The compiler generates a 'call' operation, and then (if the compilation policy decides it's required) inlines it.

At this point in code generation, all the methods called by `RuntimeEntryPoints.resolvedNewScalar(int...)` which are annotated `@Inline` are also inlined into the current method. This inlining extends through to the MMTk code so that the allocation sequence can be optimized down to a handful of instructions.

It can be instructive to look at the various levels of IR generated for object allocation using a simple test program and the `OptTestHarness` utility described elsewhere in the user guide.

16.3 Scanning Objects in Jikes RVM

One of the services that MMTk expects a virtual machine to perform on its behalf is the scanning of objects, i.e. identifying and processing the pointer fields of the live objects it encounters during collection. In principle the implementation of this interface is simple, but there are two moderately complex optimizations layered on top of this.

From MMTk's point of view, each time an object requires scanning it passes it to the VM, along with a `TransitiveClosure` object. The VM is expected to identify the pointers and invoke the `processEdge` method on each of the pointer fields in the object. The rationale for the current object scanning scheme is presented in this paper.

16.3.1 JikesRVM to MMTk Interface

MMTk requires its host virtual machine to provide an implementation of the class `org.mmtk.vm.Scanning` as its interface to scanning objects. Jikes RVM's implementation of this class is found under the source tree `MMTk/ext/vm/jikesrvm`, in the class `org.jikesrvm.mm.mmtk.Scanning`. The methods we are interested in are `scanObject(TransitiveClosure, ObjectReference)` and `specializedScanObject(int, TransitiveClosure, ObjectReference)`.

In MMTk, each plan defines one or more `TransitiveClosure` operations. Simple full-heap collectors like `MarkSweep` only define one `TransitiveClosure`, but complex plans like `GenImmix` or the `RefCount` plans define several. MMTk allows the plan to request specialized scanning on a closure-by-closure basis, closures that are specialized call `specializedScanObject` while unspecialized ones call `scanObject`. Specialization is covered in more detail below.

In the absence of hand-inlined scanning, or if specialization is globally disabled, scanning reverts to the fallback method in `org.jikesrvm.mm.mminterface.SpecializedScanMethod`. This method can be regarded as the basic underlying mechanism, and is worth understanding in detail.

```
RVMType type = ObjectModel.getObjectType(objectRef.
    toObject());
    int[] offsets = type.getReferenceOffsets();
```

This code fetches the array of offsets that Jikes RVM uses to identify the pointer fields in the object. This array is constructed by the classloader when a class is resolved.

```
if (offsets != REFARRAY_OFFSETARRAY) {
```

```

for(int i=0; i < offsets.length; i++) {
    trace.processEdge(objectRef, objectRef.toAddress().
        plus(offsets[i]));
}

```

One distinguished value (actually `null`) is used to identify arrays of reference objects, and this block of code scans scalar objects by tracing each of the fields at the offsets given by the offset array.

```

} else {
    for(int i=0; i < ObjectModel.getArrayLength(objectRef.
        toObject()); i++) {
        trace.processEdge(objectRef, objectRef.toAddress().
            plus(i << LOG.BYTES_IN_ADDRESS));
    }
}

```

The other case is reference arrays, for which we fetch the array length and scan each of the elements.

The internals of `trace.processEdge` vary by collector and by collection type (e.g. nursery/full-heap in a generational collector), and the details need not concern us here.

16.3.2 Hand Inlining

Hand inlining was introduced in February 2011, and uses a cute technique to encode 3 bits of metadata into the TIB pointer in an object's header. The 7 most frequent object patterns are encoded into these bits, and then special-case code is written for each of them.

Hand inlining produces an average-case speedup slightly better than specialization, but performs poorly on some benchmarks. This is why we use it in combination with specialization.

16.3.3 Specialized Scanning

Specialized Scanning was introduced in September 2007. It speeds up GC by removing the process of fetching and interpreting the offset array that describes each object, by jumping directly to a hard-coded method for scanning objects with a particular pattern.

The departure point from "standard" java into the specialized scanning method is `SpecializedScanMethod.invoke(...)`, which looks like this

```

@SpecializedMethodInvoke
@NoInline
public static void invoke(int id, Object object,
    TransitiveClosure trace) {
    /* By default we call a non-specialized fallback */
    fallback(object, trace);
}

```

The `@SpecializedMethodInvoke` annotation signals to the compiler that it should dispatch to one of the specialized method slots in the TIB.

Creation of specialized methods is handled by the class `org.jikesrvm.class-loader.SpecializedMethodManager`.

16.4 Using GCspy

16.4.1 The GCspy Heap Visualisation Framework

GCspy is a visualisation framework that allows developers to observe the behaviour of the heap and related data structures. For details of the GCspy model, see GCspy: An adaptable heap visualisation framework by Tony Printezis and Richard Jones, OOPSLA'02. The framework comprises two components that communicate across a socket: a *client* and a *server* incorporated into the virtual machine of the system being visualised. The client is usually a visualiser (written in Java) but the framework also provides other tools (for example, to store traces in a compressed file). The GCspy server implementation for Jikes RVM was contributed by Richard Jones of the University of Kent.

GCspy is designed to be independent of the target system. Instead, it requires the GC developer to describe their system in terms of four GCspy abstractions: *spaces*, *streams*, *tiles* and *events*. This description is transmitted to the visualiser when it connects to the server.

A *space* is an abstraction of a component of the system; it may represent a memory region, a free-list, a remembered-set or whatever. Each space is divided into a number of blocks which are represented by the visualiser as *tiles*. Each space will have a number of attributes – *streams* – such as the amount of space used, the number of objects it contains, the length of a free-list and so on.

In order to instrument a Jikes RVM collector with GCspy:

1. Provide a `startGCspyServer` method in that collector's plan. That method initialises the GCspy server with the port on which to communicate and a list of event names, instantiates drivers for each space, and then starts the server.
2. Gather data from each space for the tiles of each stream (e.g. before, during and after each collection).
3. Provide a driver for each space.

Space drivers handle communication between collectors and the GCspy infrastructure by mapping information collected by the memory manager to the space's streams. A typical space driver will:

1. Create a GCspy space.
2. Create a stream for each attribute of the space.
3. Update the tile statistics as the memory manager passes it information.
4. Send the tile data along with any summary or control information to the visualiser.

The Jikes RVM `SSGCspy` plan gives an example of how to instrument a collector. It provides GCspy spaces, streams and drivers for the semi-spaces, the immortal space and the large object space, and also illustrates how performance may be traded for the gathering of more detailed information.

16.4.2 Installation of GCspy with Jikes RVM

Building GCspy

The GCspy client code makes use of the Java Advanced Imaging (JAI) API. The build system will attempt to download and install the JAI component when required but this is only supported on the ia32-linux platform. The build system will also attempt to download and install the GCspy server when required.

Building Jikes RVM to use GCspy

To build the Jikes RVM with GCspy support the configuration parameter `config.include.gcspy` must be set to `true` such as in the `BaseBaseSemiSpaceGCspy` configuration. You can also have the Jikes RVM build process create a script to start the GCspy client tool if GCspy was built with support for client component. To achieve this the configuration parameter `config.include.gcspy-client` must be set to `true`.

The following steps build the Jikes RVM with support for GCspy on linux-ia32 platform.

```
$ cd $RVMROOT
$ ant -Dhost.name=ia32-linux -Dconfig.name=
    BaseBaseSemiSpaceGCspy -Dconfig.include.gcspy-client=1
```

It is also possible to build the Jikes RVM with GCspy support but link it against a fake stub implementation rather than the real GCspy implementation. This is achieved by setting the configuration parameter `config.include.gcspy-stub` to `true`. This is used in the nightly testing process.

Running Jikes RVM with GCspy

To start Jikes RVM with GCspy enabled you need to specify the port the GCspy server will listen on.

```
$ cd $RVMROOT/dist/BaseBaseSemiSpaceGCspy_ia32-linux
$ ./rvm -Xms20m -X:gc:gcspyPort=3000 -X:gc:gcspyWait=true
&
```

Then you need to start the GCspy visualiser client.

```
$ cd $RVMROOT/dist/BaseBaseSemiSpaceGCspy_ia32-linux
$ ./tools/gcspy/gcspy
```

After this you can specify the port and host to connect to (i.e. localhost:3000) and click the "Connect" button in the bottom right-hand corner of the visualiser.

16.4.3 Command line arguments

Additional GCspy-related arguments to the `rvm` command:

- `-X:gc:gcspyPort=<port>`

The number of the port on which to connect to the visualiser. The default is port 0, which signifies no connection.

- `-X:gc:gcs pyWait=<true|false>`
Whether Jikes RVM should wait for a visualiser to connect.
- `-X:gc:gcs pyTileSize=<size>`
How many KB are represented by one tile. The default value is 128.

16.4.4 Writing GCspy drivers

To instrument a new collector with GCspy, you will probably want to subclass your collector and to write new drivers for it. The following sections explain the modifications you need to make and how to write a driver. You may use `org.mmtk.plan.semispace.gcspy` and its drivers as an example.

The recommended way to instrument a Jikes RVM collector with GCspy is to create a `gcspy` subdirectory in the directory of the collector being instrumented, e.g. `MMTk/src/org/mmtk/plan/semispace/gcspy`. In that directory, we need 5 classes:

- `SSGCspy`,
- `SSGCspyCollector`,
- `SSGCspyConstraints`,
- `SSGCspyMutator` and
- `SSGCspyTraceLocal`.

`SSGCspy` is the plan for the instrumented collector. It is a subclass of `SS`.

`SSGCspyConstraints` extends `SSConstraints` to provide methods `boolean need-sLinearScan()` and `boolean withGCspy()`, both of which return `true`.

`SSGCspyTraceLocal` extends `SSTraceLocal` to override methods `traceObject` and `willNotMove` to ensure that tracing deals properly with GCspy objects: the `GCspyTraceLocal` file will be similar for any instrumented collector.

The instrumented collector, `SSGCspyCollector`, extends `SSCollector`. It needs to override `collectionPhase`.

Similarly, `SSGCspyMutator` extends `SSMutator` and must also override its parent's methods `collectionPhase`, to allow the allocators to collect data; and its `alloc` and `postAlloc` methods to allocate GCspy objects in GCspy's heap space.

The Plan

`SSGCspy.startGCspyServer` is called immediately before the "main" method is loaded and run. It initialises the GCspy server with the port on which to communicate, adds event names, instantiates a driver for each space, and then starts the server, forcing the VM to wait for a GCspy to connect if necessary. This method has the following responsibilities.

- Initialise the GCspy server: `server.init(name, portNumber, verbose);`
- Add each event to the ServerInterpreter ('server' for short) `server.add-Event(eventID, eventName);`

- Set some general information about the server (e.g. name of the collector, build, etc) `server.setGeneralInfo(info);`
- Create new drivers for each component to be visualised `myDriver = new MyDriver(server, args...);`

Drivers extend `AbstractDriver` and register their space with the `ServerInterpreter`. In addition to the server, drivers will take as arguments the name of the space, the MMTk space, the tilesize, and whether this space is to be the main space in the visualiser.

The Collector and Mutator

Instrumenters will typically want to add data collection points before, during and after a collection by overriding `collectionPhase` in `SSGCspyCollector` and `SSGCspyMutator`.

`SSGCspyCollector` deals with the data in the semi-spaces that has been allocated there (copied) by the collector. It only does any real work at the end of the collector's last tracing phase, `FORWARD_FINALIZABLE`.

`SSGCspyMutator` is more complex: as well as gathering data for objects that it allocated in From-space at the start of the `PREPARE_MUTATOR` phase, it also deals with the immortal and large object spaces.

At a collection point, the collector or mutator will typically

1. Return if the GCspy port number is 0 (as no client can be connected).
2. Check whether the server is connected at this event. If so, the compensation timer (which discounts the time taken by GCspy to ather the data) should be started before gathering data and stopped after it.
3. After gathering the data, have each driver call its transmit method.
4. `SSGCspyCollector` does not call the GCspy server's `serverSafePoint` method, as the collector phase is usually followed by a mutator phase. Instead, `serverSafePoint` can be called by `SSGCspyMutator` to indicate that this is a point at which the server can pause, play one event, etc.

Gathering data will vary from MMTk space to space. It will typically be necessary to resize a space before gathering data. For a space,

- We may need to reset the GCspy driver's data depending on the collection phase.
- We will pass the driver as a call-back to the allocator. The allocator will typically ask the driver to set the range of addresses from which we want to gather data, using the driver's `setRange` method. The driver should then iterate through its MMTk space, passing a reference to each object found to the driver's `scan` method.

The Driver

GCspy space drivers extend `AbstractDriver`. This class creates a new GCspy `ServerSpace` and initializes the control values for each tile in the space. Control values indicate whether a tile is *used*, *unused*, a *background*, a *separator* or a *link*. The constructor for a typical space driver will:

1. Create a GCspy Stream for each attribute of a space.
2. Initialise the tile statistics in each stream.

Some drivers may also create a `LinearScan` object to handle call-backs from the VM as it sweeps the heap (see above).

The chief roles of a driver are to accumulate tile statistics, and to transmit the summary and control data and the data for all of their streams. Their data gathering interface is the `scan` method (to which an object reference or address is passed).

When the collector or mutator has finished gathering data, it calls the `transmit` of the driver for each space that needs to send its data. Streams may send values of types `byte`, `\spverbshort+` or `int`, implemented through classes `ByteStream`, `ShortStream` or `IntStream`. A driver's `transmit` method will typically:

1. Determine whether a GCspy client is connected and interested in this event, e.g. `server.isConnected(event)`
2. Setup the summaries for each stream, e.g. `stream.setSummary(values...);`
3. Setup the control information for each tile. e.g.

```
controlValues(CONTROL_USED, start, numBlocks);
controlValues(CONTROL_UNUSED, end, remainingBlocks);
```

4. Set up the space information, e.g. `setSpace(info);`
5. Send the data for all streams, e.g. `send(event, numTiles);`

Note that `AbstractDriver.send` takes care of sending the information for all streams (including control data).

Subspaces

Subspace provides a useful abstraction of a contiguous region of a heap, recording its start and end address, the index of its first block, the size of blocks in this space and the number of blocks in the region. In particular, `Subspace` provides methods to:

- Determine whether an address falls within a subspace;
- Determine the block index of the address;
- Calculate how much space remains in a block after a given address;

Part III

MMTk Tutorial

This tutorial will build up a sophisticated garbage collector from scratch, starting with the empty shell that is the NoGC "collector" in MMTk (collector is a misnomer in this case since NoGC does not collect), and gradually adding functionality.

This tutorial will tell you the mechanics of building a collector in MMTk. It will tell you how but it does not tell you anything about why. The tutorial thus serves two purposes: 1) to give you some insight into the mechanics of MMTk (but not the underlying reasons or design rationale), and 2) show you that the mechanics of building a non-trivial GC in MMTk is not hard, hopefully giving you confidence to start exploring MMTk more deeply.

Please use the latest release to work with the tutorial. If you run into trouble following the instructions, please try again with `git HEAD`. If the problem persists, please report this as a bug.

1. Preliminaries
2. Building a Mark-sweep Collector
3. Building a Hybrid Copying/Mark-Sweep Collector

Chapter 17

Preliminaries

17.1 Getting MMTk and Jikes RVM and Eclipse working

1. Download the latest Jikes RVM release (or use git HEAD)
2. Ensure you can Build and Run Jikes RVM.
3. Ensure you can build and run the BaseBaseNoGC configuration (build with: `bin/buildit localhost BaseBaseNoGC`, run with something like:

```
dist/BaseBaseNoGC_ia32-linux/rvm HelloWorld
```

Note that this configuration *does not* perform garbage collection so can only run small benchmarks which do not exhaust available memory. This configuration will be used as the basis for the tutorial.

4. Ensure that your source is successfully imported (and editable) within an IDE such as Eclipse.
5. Set up an Eclipse Run configuration for the NoGC plan using the MMTk Test Harness.

17.2 Creating The Base Tutorial Collector

1. Copy the `org.mmtk.plan.nogc` package to `org.mmtk.plan.tutorial` (copy and paste the package in Eclipse).
2. Rename the constituent classes from `NoGC*` to `Tutorial*` (use Refactor → Rename on each class within the `org.mmtk.plan.tutorial` package in Eclipse).
3. Edit file class `org.mmtk.harness.PlanSpecificConfig`, and add the following lines

```
register(new PlanSpecific("org.mmtk.plan.tutorial.  
Tutorial").addExpectedSpaces("default"), "  
Tutorial");
```

to the static initializer (look for "NoGC").

4. Modify your MMTk Harness Eclipse Run Configuration to use the new Plan (the name of the plan is the second parameter to the "register" method you inserted above), and click 'Run' to run it.
5. Build and run the resulting collector.
Build with something like:

```
bin/buildit localhost BaseBaseTutorial
```

run with something like:

```
dist/BaseBaseTutorial_ia32 - linux/rvm HelloWorld
```

Chapter 18

Building a Mark-sweep Collector

We will now modify the `Tutorial` collector to perform allocation and collection according to a mark-sweep policy. First we will change the allocation from bump-allocation to free-list allocation (but still no collector whatsoever), and then we will add a mark-sweep collection policy, yielding a complete mark-sweep collector.

18.1 Free-list Allocation

This step will change your simple collector from using a bump pointer to a free list (but still without any garbage collection).

1. Update the constraints for this collector to reflect the constraints of a mark-sweep system, by updating `TutorialConstraints` as follows:
 - `gcHeaderBits()` should return `MarkSweepSpace.LOCAL_GC_BITS_REQUIRED`.
 - `gcHeaderWords()` should return `MarkSweepSpace.GC_HEADER_WORDS_REQUIRED`.
 - The `maxNonLOSDefaultAllocBytes()\spverb` method should be added, overriding one provided by the base class, and should return `SegregatedFreeListSpace.MAX_FREELIST_OBJECT_BYTES` (because this reflects the largest object size that can be allocated with the free list allocator).
2. In `Tutorial`, replace the `ImmortalSpace` with a `MarkSweepSpace`:
 - (a) rename the variable `noGCspace` to `msSpace` (right-click, Refactor → Rename...)
 - (b) rename the variable `NOGC` to `MARK_SWEEP` (right-click, Refactor → Rename...)
 - (c) change the type and static initialization of `msSpace` to

```
public static final MarkSweepSpace msSpace = new
    MarkSweepSpace("ms", VMRequest.discontiguous()
    );
```

- (d) add an import for `MarkSweepSpace` and remove the redundant import for `ImmortalSpace`.
3. In `TutorialMutator`, replace the `ImmortalLocal` (a bump pointer) with a `MarkSweepLocal` (a free-list allocator)
 - (a) change the type of `nogc` and change the static initializer appropriately.
 - (b) change the appropriate import statement from `ImmortalLocal` to `MarkSweepLocal`.
 - (c) rename the variable `nogc` to `ms` (right-click, Refactor → Rename...)
4. Also in `TutorialMutator`, fix `postAlloc()` to initialize the mark-sweep header:

```
if (allocator == Tutorial.ALLOC_DEFAULT) {
    Tutorial.msSpace.postAlloc(ref);
} else {
    super.postAlloc(ref, typeRef, bytes, allocator);
}
```

5. In `PlanSpecificConfig`, find the line for `Tutorial`, and change "default" to "ms"

With these changes, `Tutorial` should now work, just as it did before, only exercising a free list (mark-sweep) allocator rather than a bump pointer (immortal) allocator. Create a `BaseBaseTutorial` build, and test your system to ensure it performs just as it did before. You may notice that its memory is exhausted slightly earlier because the free list allocator is slightly less efficient in space utilization than the bump pointer allocator.

18.2 Mark-Sweep Collection

The next change required is to perform mark-and-sweep collection whenever the heap is exhausted. The `poll()` method of a plan is called at appropriate intervals by other MMTk components to ask the plan whether a collection is required.

1. Change `TutorialConstraints` so that it inherits constraints from a collecting plan:

```
public class TutorialConstraints extends
    StopTheWorldConstraints
```

2. The plan needs to know how to perform a garbage collection. Collections are performed in phases, coordinated by data structures defined in `StopTheWorld`, and have global and thread-local components. First ensure the global components are behaving correctly. These are defined in `Tutorial` (which is implicitly *global*).

- (a) Make `Tutorial` extend `StopTheWorld` (for stop-the-world garbage collection) rather than `Plan` (the superclass of `StopTheWorld`):

```
public class Tutorial extends StopTheWorld
```

- (b) Rename the trace variable to `msTrace` (right-click, Refactor → Rename...)
- (c) Add code to ensure that `Tutorial` performs the correct global collection phases in `collectionPhase()`:

- i. First remove the assertion that the code is never called (`if (VM.VERIFY_ASSERTIONS) VM.assertions._assert(false);`).
- ii. Add the prepare phase, preparing both the global tracer (`msTrace`) and the space (`msSpace`), after first performing the preparation phases associated with the superclasses. Using the commented template in `Tutorial.collectionPhase()`, set the following within the clause for `phaseId == PREPARE`:

```
if (phaseId == PREPARE) {
    super.collectionPhase(phaseId);
    msTrace.prepare();
    msSpace.prepare(true);
    return;
}
```

- iii. Add the closure phase, again preparing the global tracer (`msTrace`):

```
if (phaseId == CLOSURE) {
    msTrace.prepare();
    return;
}
```

- iv. Add the release phase, releasing the global tracer (`msTrace`) and the space (`msSpace`) before performing the release phases associated with the superclass:

```
if (phaseId == RELEASE) {
    msTrace.release();
    msSpace.release();
    super.collectionPhase(phaseId);
    return;
}
```

- v. Finally ensure that for all other cases, the phases are delegated to the superclass, uncommenting the following after all of the above conditionals:

```
super.collectionPhase(phaseId);
```

- (d) Add a new accounting method that determines how much space a collection needs to yield to the mutator. The method, `getPagesUsed`, overrides the one provided in the `StopTheWorld` superclass:

```
@Override
public int getPagesUsed() {
    return super.getPagesUsed() + msSpace.
        getPagesUsed();
}
```

- (e) Add a new method that determines whether an object will move during collection:

```
@Override
public boolean willNeverMove(ObjectReference
    object) {
    if (Space.isInSpace(MARKSWEEP, object))
        return true;
    return super.willNeverMove(object);
}
```

3. Next ensure that Tutorial correctly performs *local* collection phases. These are defined in `TutorialCollector`.

- (a) Make `TutorialCollector` extend `StopTheWorldCollector`:

- i. Extend the class (`public class TutorialCollector extends StopTheWorldCollector`).
- ii. Import `StopTheWorldCollector`.
- iii. Remove some methods now implemented by `StopTheWorldCollector`: `collect()` and (if present) `concurrentCollect()` and `concurrentCollectionPhase()`.

- (b) Add code to ensure that `Tutorial` performs the correct global collection phases in `collectionPhase()`:

- i. First remove the assertion that the code is never called (`if (VM.VERIFY_ASSERTIONS) VM.assertions._assert(false);`).
- ii. Add the prepare phase, preparing both the local tracer (`trace`) after first performing the preparation phases associated with the superclasses. Using the commented template in `TutorialCollector.collectionPhase()`, set the following within the clause for `phaseId == PREPARE`:

```
if (phaseId == Tutorial.PREPARE) {
    super.collectionPhase(phaseId, primary);
    trace.prepare();
    return;
}
```

- iii. Add the closure phase, completing the local tracer (**trace**):

```
if (phaseId == Tutorial.CLOSURE) {
    trace.completeTrace();
    return;
}
```

- iv. Add the release phase, releasing the local tracer (**trace**) before performing the release phases associated with the superclass:

```
if (phaseId == Tutorial.RELEASE) {
    trace.release();
    super.collectionPhase(phaseId, primary);
    return;
}
```

- v. Finally ensure that for all other cases, the phases are delegated to the superclass, uncommenting the following after all of the above conditionals:

```
super.collectionPhase(phaseId, primary);
```

4. Finally ensure that **Tutorial** correctly performs local mutator-related collection activities:

- (a) Make **TutorialMutator** extend **StopTheWorldMutator**:

- i. Extend the class: `public class TutorialMutator extends StopTheWorldMutator`.
- ii. Import `StopTheWorldMutator`.

- (b) Update the mutator-side collection phases:

- i. First remove the assertion that the code is never called (`if (VM.VERIFY_ASSERTIONS) VM.assertions._assert(false);`).
- ii. Add the prepare phase to `collectionPhase()` which prepares mutator-side data structures (namely the per-thread free lists) for the start of a collection:

```
if (phaseId == Tutorial.PREPARE) {
    super.collectionPhase(phaseId, primary);
    ms.prepare();
    return;
}
```

- iii. Add the release phase to `collectionPhase()` which re-initializes mutator-side data structures (namely the per-thread free lists) after the end of a collection:

```
if (phaseId == Tutorial.RELEASE) {
    ms.release();
    super.collectionPhase(phaseId, primary);
    return;
}
```


- iv. Finally, delegate all other phases to the superclass:

```
super.collectionPhase(phaseId, primary);
```

With these changes, `Tutorial` should now work with both mark-sweep allocation *and* collection. Create a `BaseBaseTutorial` build, and test your system to ensure it performs just as it did before. You can observe the effect of garbage collection as the program runs by adding `-X:gc:verbose=1` to your command line as the first argument after `rvm`. If you run a very simple program (such as `HelloWorld`), you might not observe any garbage collection. In that case, try running a larger program such as a DaCapo benchmark. You may also observe that the output from `-X:gc:verbose=1` indicates that the heap is growing. Dynamic heap resizing is normal default behavior for a JVM. You can override this by providing minimum (`-Xms`) and maximum (`-Xmx`) heap sizes (these are standard arguments respected by all JVMs. The heap size should be specified in bytes as an integer and a unit (K, M, G), for example: `-Xms20M -Xmx20M`).

18.3 Optimized Mark-sweep Collection

MMTk has a unique capacity to allow specialization of the performance-critical scanning loop. This is particularly valuable in collectors which have more than one mode of collection (such as in a generational collector), so each of the collection paths is explicitly specialized at build time, removing conditionals from the hot portion of the tracing loop at the core of the collector. Enabling this involves just two small steps:

1. Indicate the number of specialized scanning loops and give each a symbolic name, which at this stage is just one since we have a very simple collector:

- (a) Override the `numSpecializedScans()` getter method in `TutorialConstraints`:

```
@Override
public int numSpecializedScans() { return 1; }
```

- (b) Define a constant to represent our (only) specialized scan in `Tutorial` (we will call this scan "mark"):

```
public static final int SCAN_MARK = 0;
```

2. Register the specialized method:

- (a) Add the following line to `registerSpecializedMethods()` method in `Tutorial`:

```
TransitiveClosure.registerSpecializedScan(
    SCAN_MARK, TutorialTraceLocal.class);
```

- (b) Add `Tutorial.SCAN_MARK` as the first argument to the superclass constructor for `TutorialTraceLocal`:

```
public TutorialTraceLocal(Trace trace) {  
    super(Tutorial.SCANMARK, trace);  
}
```

Chapter 19

Building a Hybrid Collector

Extend the Tutorial plan to create a "copy-MS" collector, which allocates into a copying nursery and at collection time, copies nursery survivors into a mark-sweep space. This plan does not require a write barrier (it is not strictly generational, as it will collect the whole heap each time the heap is full). Later we will extend it with a write barrier, allowing the nursery to be collected in isolation. Such a collector would be a generational mark-sweep collector, similar to GenMS.

19.1 Add a Copying Nursery

This step will change your simple collector from using a bump pointer to a free list (but still without any garbage collection).

1. In `TutorialConstraints`, make the following changes:

- (a) Override the `movesObjects()` method to return `true`, reflecting that we are now building a copying collector:

```
@Override
public boolean movesObjects() { return true; }
```

- (b) Remove the restriction on default alloc bytes (since default allocation will now go to a bump-pointed space). To do this, remove the override of `maxNonLOSDefaultAllocBytes()`.
- (c) Add a restriction on the maximum size that may be copied into the (default) non-LOS mature space:

```
@Override
public int maxNonLOSCopyBytes() { return
    SegregatedFreeListSpace.
    MAX_FREELIST_OBJECT_BYTES; }
```

2. In `Tutorial`, add a nursery space:

- (a) Create a new space, `nurserySpace`, of type `CopySpace`. The new space will initially be a from-space, so provide `false` as the third argument. Initialize the space with a contiguous virtual memory region consuming 0.15 of the heap by passing "0.15" and "true" as arguments to the constructor of `VMRequest` (more on this later). Create and initialize a new integer constant to hold the descriptor for this new space:

```
public static final CopySpace nurserySpace = new
    CopySpace("nursery", false, VMRequest.
        highFraction(0.15f));
public static final int NURSERY = nurserySpace.
    getDescriptor();
```

- (b) Add the necessary import statements
- (c) Add `nurserySpace` to the `PREPARE` and `RELEASE` phases of `collectionPhase()`, prior to the existing calls to `msTrace`. Pass true to `nurserySpace.prepare()` indicating that the nursery is a *from-space* during collection.
- (d) Fix accounting so that `Tutorial` accounts for space consumed by `nurserySpace`:
- Add `nurserySpace` to the equation in `getPagesUsed()`
- (e) Since initial allocation will be into a copying space, we need to account for copy reserve:
- Add a method to override `getCollectionReserve()` which returns `nurserySpace.reservedPages() + super.getCollectionReserve()`
 - Add a method to override `getPagesAvail()`, returning `getTotalPages() - getPagesReserved() >> 1;`

19.2 Add nursery allocation

In `TutorialMutator`, replace the free-list allocator (`MarkSweepLocal`) with a nursery allocator: Use an instance of `CopyLocal`, calling it `nursery`. The constructor argument should be `Tutorial.nurserySpace`:

- change `alloc()` to use `nursery.alloc()` rather than `ms.alloc()`.
- remove the call to `msSpace.postAlloc()` from `postAlloc()` since there is no special post-allocation work necessary for the new copy space. The call to `super.postAlloc()` should remain conditional on `allocator != Tutorial.ALLOC_DEFAULT`.
- change the check within `getAllocatorFromSpace()` to check against `Tutorial.nurserySpace` and to return `nursery`.
- adjust `collectionPhase`
 - replace call to `ms.prepare()` with `nursery.reset()`
 - remove call to `ms.release()` since there are no actions necessary for the nursery allocator upon release.

19.3 Add copying to the collector

In `TutorialCollector` add the capacity for the collector to allocate (copy), since our new hybrid collector will perform copying.

1. Add local allocators for both large object space and the mature space:

```
private final LargeObjectLocal los = new
    LargeObjectLocal(Plan.loSpace);
private final MarkSweepLocal mature = new
    MarkSweepLocal(Tutorial.msSpace);
```

2. Add an `allocCopy()` method that conditionally allocates to the LOS or mature space:

```
@Override
public final Address allocCopy(ObjectReference
    original, int bytes,
                                int align, int offset,
                                int allocator) {
    if (allocator == Plan.ALLOCLOS)
        return los.alloc(bytes, align, offset);
    else
        return mature.alloc(bytes, align, offset);
}
```

3. Add a `postCopy()` method that conditionally calls LOS or mature space post-copy actions:

```
@Override
public final void postCopy(ObjectReference object,
    ObjectReference typeRef,
                                int bytes, int allocator)
    {
    if (allocator == Plan.ALLOCLOS)
        Plan.loSpace.initializeHeader(object, false);
    else
        Tutorial.msSpace.postCopy(object, true);
    }
```

19.4 Make necessary changes to `TutorialTraceLocal`

1. Add nurserySpace clauses to `isLive()` and `traceObject()`:

- (a) Add the following to `isLive()`:

```
if (Space.isInSpace(Tutorial.NURSERY, object))
    return Tutorial.nurserySpace.isLive(object);
```

- (b) Add the following to `traceObject()`:

```
if (Space.isInSpace(Tutorial.NURSERY, object))  
    return Tutorial.nurserySpace.traceObject(this,  
        object, Tutorial.ALLOCDEFAULT);
```

2. Add a new `willNotMoveInCurrentCollection()` method, which identifies those objects which do not move (necessary for copying collectors):

```
@Override  
public boolean willNotMoveInCurrentCollection(  
    ObjectReference object) {  
    return !Space.isInSpace(Tutorial.NURSERY, object);  
}
```

3. Modify `PlanSpecificConfig` to add the new `nursery` space:

```
new PlanSpecific("org.mmtk.plan.tutorial.Tutorial").  
    addExpectedSpaces("ms", "nursery"), "Tutorial");
```

With these changes, `Tutorial` should now work. You should be able to again build a `BaseBaseTutorial` image and test it against any benchmark. Again, if you use `-X:gc:verbose=3` you can see the movement of data among the spaces at each garbage collection.