

Space Details

Key:	RVM
Name:	RVM
Description:	The Jikes Research Virtual Machine (RVM) is designed to execute Java(TM) programs and is typically used in research on fundamental virtual machine design issues.
Creator (Creation Date):	xircles (Jan 19, 2007)
Last Modifier (Mod. Date):	pdonald (Jan 20, 2007)

Available Pages

- User Guide
 - Architecture
 - Adaptive Optimization System
 - Compilers
 - Baseline Compiler
 - JNI Compiler
 - Optimizing Compiler
 - BURS
 - Compiler Optimization Comparison Chart
 - IR
 - OptTestHarness
 - Core Runtime Services
 - Bootstrap
 - Class and Code Management
 - Exception Management
 - JNI
 - Object Model
 - Thread Management
 - VM Callbacks
 - VM Conventions
 - Magic
 - Compiler Intrinsics
 - Raw Memory Access
 - Unboxed Types
 - Uninterruptible Code
 - MMTk
 - Using GCSPy
 - Care and Feeding
 - Building the RVM
 - Building Patched Versions

- Cross-Platform Building
- Primordial Class List
- Configuring the RVM
- Debugging the RVM
- Experimental Guidelines
- Get The Source
- Modifying the RVM
 - Coding Conventions
 - Coding Style
- Profiling Applications with Jikes RVM
- Quick Start Guide
- Running the RVM
- Testing the RVM
 - External Test Resources

User Guide

This page last changed on Mar 07, 2007 by [pdonald](#).

The User Guide provides Jikes™ 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 the 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 mailing list. Constructive criticism will be cheerfully accepted.

- [Care and Feeding](#): The guide to practical aspects of building, testing, debugging and evaluating the Jikes RVM.
- [Architecture](#): The guide to the major architectural decisions of the Jikes RVM.

Architecture

This page last changed on Mar 08, 2007 by [pdonald](#).

This section describes the architecture of the Jikes RVM. The 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.
- [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.

Adaptive Optimization System

This page last changed on Mar 08, 2007 by [dgrove](#).

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.

Compilers

This page last changed on Mar 09, 2007 by [pdonald](#).

- [Baseline Compiler](#)
- [JNI Compiler](#)
- [Optimizing Compiler](#)

Baseline Compiler

This page last changed on Mar 07, 2007 by [pdonald](#).

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 `VM_BaselineCompiler`. The main platform-dependent file is `VM_Compiler`.

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

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.

Command-Line Options

The command-line options to the baseline compiler are stored as fields in an object of type `VM_BaselineOptions`; this file is mechanically generated by the build process. To add or modify the command-line options in `VM_BaselineOptions.java`, you must modify either `BooleanOptions.dat`, or `ValueOptions.dat`. You should describe your desired command-line option in a format described below in the appendix; you will also find the details for the optimizing compiler's command-line options. Some options are common to both the baseline compiler and optimizing compiler. They are defined by the `SharedBooleanOptions.dat` and `SharedValueOptions.dat` files found in the `rvm/src-generated/options` directory.

JNI Compiler

This page last changed on Jan 24, 2007 by [dgrove](#).

The JNI compiler "compiles" native methods by generating code to transition from Jikes RVM internal calling/register conventions to the native platforms ABI.

See also THE JNI IMPL DETAILS section.

Optimizing Compiler

This page last changed on Mar 09, 2007 by [pdonald](#).

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

- [Method Compilation](#): The fundamental unit for compilation in the 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.
- [Compiler Optimization Comparison Chart](#): Chart comparing the Jikes RVM optimizing compiler to compilers in other JVMs.

BURS

This page last changed on Mar 09, 2007 by [pdonald](#).

The optimizing compiler uses the Bottom-Up Rewrite System (BURS) for instruction selection. BURS is essentially a tree pattern matching system derived from Iburg by David R. Hanson. (See "Engineering a Simple, Efficient Code-Generator Generator" by Fraser, Hanson, and Proebsting, LOPLAS 1(3), Sept. 1992.) The instruction selection rules for each architecture are specified in an architecture-specific files 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.

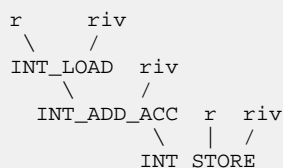
Each rule is defined by a four-line record, consisting of:

- **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 Java™ 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 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).

What the above rule basically reads is the following:

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

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

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

Future directions

Whilst jburg allows us to do good instruction selection there are a number of areas where it is lacking:

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): ADD(r,r), 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 with a modified BURS system they can achieve this result. Their syntax is:

```
r: ADD(r,r)
r: ADD(r,r)
```

- [Rainer Leupers, Code selection for media processors with SIMD instructions, 2000](#)

Compiler Optimization Comparison Chart

This page last changed on Mar 07, 2007 by [pdonald](#).

This section presents a comparison chart of the Jikes RVM against the following JVMs.

- [IBM JDK v6 r0](#)

Inlining

Category	Optimization	RVM	IBM JDK v6 r0
Inlining	Trivial Inlining	✓	✓
	Call graph inlining	?	✓
	Tail recursion elimination	✓	✓
	Virtual call guard optimizations	?	✓
Local optimizations	Local data flow analyses and optimization	✓	✓
	Register usage optimization	✓	✓
	Simplification of Java idioms	✓	✓
Control flow optimizations	Code reordering, splitting and removal	✓	✓
	Loop reduction and inversion	✗	✓
	Loop invariant code motion	✓	✓
	Loop striding	✗	✓
	Loop unrolling	✓	✓
	Loop peeling	✗	✓
	Loop versioning	✓ (disabled)	✓
	Loop specialization	✗	✓
	Exception directed optimization	?	✓
	Switch analysis	?	✓

Global optimizations	Global flow analyses and optimization	?	✓
	Partial redundancy elimination	?	✓
	Escape analysis	?	✓
	GC and memory allocation optimizations	?	✓
	Synchronization optimizations	?	✓
Native code generation	Small optimization based on architecture characteristics	?	✓

IR

This page last changed on Mar 09, 2007 by [pdonald](#).

The optimizing compiler intermediate representation (IR) is held in an object of type `OPT_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 `OPT_IR`.

This section documents basic information about the intermediate instruction. For a tutorial based introduction to the material it is highly recommended that you read "[Jikes RVM Optimizing Compiler Intermediate Code Representation](#)".

IR Operators

The IR operators are defined by the class `OPT_Operators`, which in turn is automatically generated from a template by a driver. The input to the driver are two files, both called `OperatorList.dat`. One input file resides in `$RVM_ROOT/rvm/src-generated/opt-ir` and defines machine-independent operators. The other resides in `$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 `OPT_Operator.java` for a list of valid traits.
- **IMPLDEFS:** set of registers implicitly defined by this operator; usually applies only to machine-dependent operators
- **IMPLUSES:** set of registers implicitly used by this operator; usually applies only to machine-dependent operators

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

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

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

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

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

the assembler. See the file for details.

Instruction Formats

Every IR instruction fits one of the pre-defined *Instruction Formats*. The Java package `com.ibm.jikesrvm.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 `OPT_Operands` interface and the `Move` instruction format:

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

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

Each fixed-length instruction format is defined in the text file

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

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

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

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

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

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

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

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

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

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

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

OptTestHarness

This page last changed on Mar 09, 2007 by [pdonald](#).

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 `org.jikesrvm.tools.oth.OptTestHarness` program provides even more control over the optimizing compiler. This driver program allows you to invoke the optimizing compiler as an "application" running on top of the VM.

Command Line Options

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

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.

Core Runtime Services

This page last changed on Mar 11, 2007 by [pdonald](#).

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.
- [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.

Bootstrap

This page last changed on Mar 11, 2007 by [pdonald](#).

The RVM is started up by a boot program written in C. This program is responsible for

- registering signal handlers to deal with the hardware errors generated by the RVM
- establishing the initial virtual memory map employed by the RVM
- mapping the 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 the RVM image area
- setting up the JTOC and PR registers for its `VM_Processor`/pthread
- 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 the 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 and processor 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 Classpath 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 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 then enables the Java scheduler, creating extra pthreads to support the required number of virtual processors (VPs) and rendezvousing with them before letting them idle waiting for new Java threads. The scheduler boot routine places an idle thread `VM_IdleThread` and garbage collector thread `VM_CollectorThread` into the idle queue and collector queue.

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.

Memory Map

The 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.

RVM Image Segment

The next area is the one initialized by the boot program to contain the 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 processor instance on which to start Java execution
- 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.

RVM Heap Segment

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

Class and Code Management

This page last changed on Mar 10, 2007 by [pdonald](#).

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 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.

Class Loading

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

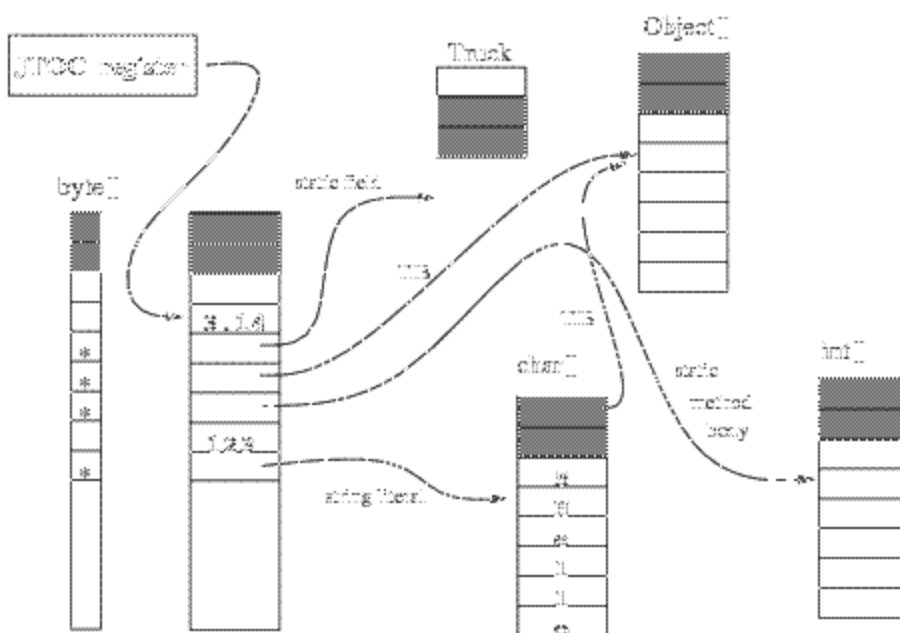
- **vacant:** The VM_Class object for this class has been created and registered and is in the process of being loaded.
- **loaded:** The class's bytecode file has been read and parsed successfully. The modifiers and attributes for the fields have been loaded and the constant pool has been constructed. 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.

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 `VM_InterfaceInvocation`.

Exception Management

This page last changed on Mar 11, 2007 by [pdonald](#).

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 except on AIX and this results in faster code generation for non-exceptioning code.

The 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 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 RVM booter program 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 `VM_Runtime.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 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 `VM_Runtime.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.

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

JNI Functions

All of the 1.1 through 1.4 `JNIEnv` interface functions are implemented.

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

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 `VM_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, the RVM compilers generate a call sequence consistent with the platform's underlying calling convention. A syscall is not a GC-safe point, so syscalls may modify the Java heap (eg. `memcpy()`). For more details on the mechanics of adding a new syscall to the system, see the header comments of `VM_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 `VM_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.

Interactions with *m-to-n* Threading

See the [Thread Management](#) subsection for more details on the thread system and *m-to-n* threading in Jikes RVM.

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

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

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

On Linux/x86, Jikes RVM "hijacks" certain blocking system calls and reflects them back into the VM. The VM then uses nonblocking equivalents. This handles many of the common cases of blocking native code without requiring the full complexity of the timer-based preemption mechanism that we used to use on AIX. A complete solution would consist of implementing both mechanisms on both platforms. We hope to do this in the future.

We got GNU Classpath's (JNI-based) AWT support to work by adding code to Classpath that tells GTK (the windowing toolkit Classpath uses) to use Jikes RVM's Java threading primitives instead of the pthread-based ones that it uses by default. Jikes RVM automatically tells Classpath to do this by setting the Java system property `gnu.classpath.awt.gtk.portable.native.sync` at boot time. If your native code uses the `glib` threading primitives, as GTK does, then this will work for you, too.

Implementation Details

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

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

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.
- **VM_JNICompiler:** The only known deficiency in `VM_JNICompiler` 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.

- **Directly-Exported Invocation Interface Functions:** These functions (`GetDefaultJavaVMInitArgs`, `JNI_CreateJavaVM`, and `JNI_GetCreatedJavaVMs`) are 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.

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

Object Model

This page last changed on Mar 08, 2007 by [pdonald](#).

Object Model

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

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. However, the AIX™ operating system permits loads from low memory, but accesses to very high memory (at small *negative* offsets from a null pointer) normally cause hardware interrupts. Therefore on AIX only a subset of pointer dereferences can be protected by an implicit null check.

Object Header

Logically, every object header contains the following components:

- **TIB Pointer:** The TIB (Type Information Block) holds information that applies to all objects of a type. The structure of the TIB is defined by `VM_TIBLayoutConstants`. 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.

- **Hash Code:** Each Java object has an identity hash code. This can be read by *Object.hashCode* or in the case that this method 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.
- **Lock:** Each Java object has an associated lock state. This could be a pointer to a lock object or a direct representation of the lock.
- **Array Length:** Every array object provides a length field that contains the length (number of elements) of the array.
- **Garbage Collection Information:** Each Java object has associated information used by the memory management system. Usually this consists of one or two mark bits, but this could also include some combination of a reference count, forwarding pointer, etc.
- **Misc Fields:** In experimental configurations, the object header can be expanded to add additional fields to every object, typically to support profiling.

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

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.

Double and long fields benefit from being 8 byte aligned. Every *VM_Class* 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 4bytes 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.

Thread Management

This page last changed on Mar 11, 2007 by [pdonald](#).

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 `VM_Thread`. These threads are multiplexed onto one or more virtual processors (see `VM_Processor`). The number of Jikes RVM virtual processors to use is a command line argument (e.g. `-X:processors=4`). If no command line argument is given, Jikes RVM will default to creating only one virtual processor. If you want Jikes RVM to utilize more than 1 CPU, then you need to tell it to use the appropriate number of virtual processors. Multiple virtual processors require a working pthread library, each virtual processor being bound to a pthread.

The Jikes RVM is a M:N thread model, scheduling execution of an arbitrarily large number (M) of Java threads over a finite number (N) of `VM_Processors`. This means that at most N Java threads can be executing concurrently (true concurrency requires that the underlying platform incorporate multiple execution contexts capable of running several pthreads simultaneously). For maximal performance, you should tell Jikes RVM to create one virtual processor for each CPU on an SMP.

A benefit of M:N threading is that the Java system can only obtain N pthreads worth of scheduled execution time from the underlying operating system. By contrast a 1:1 model would allow the Java system to swamp the system with runnable threads, crowding out system threads and other Unix applications (Of course, in certain situations 1:1 threading may be more appropriate). Another benefit is that system-wide thread management within the RVM involves synchronizing at most N active threads (N is an RVM-boot time constant) rather than an unbounded number of threads. For example, a stop the world garbage collector merely needs to flag the N currently active threads that they should switch into a collector thread rather than having to stop every mutator thread in the system.

A downside to the M:N threading model is that a given Java thread may be scheduled for execution by different pthreads at different stages during its execution. In particular, when the Java thread calls native methods which rely upon per-pthread storage this choice of threading model is disastrous. Unfortunately many threaded library implementations do exactly this.

Thread Queues

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

Transferring execution from one thread (A) to another (B) is a complex operation negotiated by the `yield` and `morph` methods of `VM_Thread` and the `dispatch` method of `VM_Processor`. `yield` places A on an indicated queue (releasing the lock on the queue, if it is global). `morph` does some additional housekeeping and transfers control to `dispatch` which selects the next thread to execute. `Dispatch` then invokes `VM_Magic.threadSwitch` to save the hardware context of A and restore the hardware context of B. It now appears as if B's previous call to `dispatch` has just returned and it continues executing. While

dispatching is proceeding (from the time A is enqueued until B's hardware context is restored), the `beingDispatched` field of A is set to prevent it from being scheduled for execution on some other virtual processor while it is still executing in morph or dispatch.

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

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

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

Synchronization

Jikes RVM uses a light-weight locking scheme to implement Java monitors (see `VM_Lock` and `VM_ThinLock`). The exact details of the locking scheme are dependent on which variant of `VM_JavaHeader.java` is selected at system build time. If an object instance has a light weight lock, then some bits in the object header are used for locking. If the top bit is set, the remainder of the bits are an index into an array of heavy-weight locks. Otherwise, if the object is locked, these bits contain the id of the thread that holds the lock and a count of how many times it is held. If a thread tries to lock an object locked with a light-weight lock by another thread, it can spin, yield, or inflate the lock. Spinning is probably a bad idea. The number of times to yield before inflating is a matter open for investigation (as are a number of locking issues, see `VM_Lock`). Heavy-weight locks contain an `enteringQueue` for threads trying to acquire the lock.

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

IO Management

Many native IO operations are blocking operations and the thread invoking the operation will block until the IO operation completes. This causes problems for the Jikes RVM M:N thread model. If a `VM_Thread` invokes such an operation the whole `VM_Processor` will be blocked and unable to schedule any other `VM_Threads`.

The Jikes RVM attempts to avoid this problem by intercepting blocking IO operations and replacing them with non-blocking operations. The calling thread is then suspended and placed in a `VM_ThreadIOQueue`.

The `VM_Processor` periodically checks pending IO operations and after they complete the calling thread is moved from the `VM_ThreadIOQueue` back into the running queue. The Jikes RVM may not always be able to intercept blocking IO operations in native code and can not insert yield points in native code. As a result a long running or blocked native method can block other threads.

VM Callbacks

This page last changed on Mar 08, 2007 by [pdonald](#).

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

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

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

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

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 `VM_Callbacks` interface. The specifics are specified below:

1. Modify `spec/harness/ProgramRunner.java`:
 - a. add an import statement for the `VM_Callbacks` class:

```
import com.ibm.jikesrvm.VM_Callbacks;
```

- b. before the call to `runOnce` add the following:

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

- c. after the call to `runOnce` add the following:

```
VM_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 `VM_Callbacks.java` and place it the appropriate directory structure with your modified file, i.e., `com/ibm/jikesrvm/VM_Callbacks.java`

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

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

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 `VM_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.

VM Conventions

This page last changed on Mar 08, 2007 by [pdonald](#).

AIX/PowerPC VM Conventions

This section describes register, stack, and calling conventions that apply to Jikes RVM on AIX/PowerPC [™](#).

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. In particular, we may move to the AIX [™](#) conventions at a later date. Where code differs from the AIX conventions, it should be marked with a comment to that effect containing the string "AIX".

Register conventions

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

- **Scratch:** Needed for method prologue/epilogue. Can be used by compiler between calls.
- **Dedicated:** Reserved registers with known contents:
 - **JTOC** - Jikes RVM Table Of Contents. Globally accessible data: constants, static fields and methods.
 - **FP** - Frame Pointer Current stack frame (thread specific).
 - **PR** - Processor register. An object representing the current virtual processor (the one executing on the CPU containing these registers). A field in this object contains a reference to the object representing the VM_Thread being executed.
- **Volatile ("caller save", or "parameter"):** Like scratch registers, these can be used by the compiler as temporaries, but they are not preserved across calls. Volatile registers differ from scratch registers in that volatiles can be used to pass parameters and result(s) to and from methods.
- **Nonvolatile ("callee save", or "preserved"):** These can be used (and are preserved across calls), but they must be saved on method entry and restored at method exit. Highest numbered registers are to be used first. (At least initially, nonvolatile registers will not be used to pass parameters.)
- **Condition Register's 4-bit fields:** We follow the AIX conventions to minimize cost in JNI transitions between C and Java code. The baseline compiler only uses CR0. The opt compiler allocates CR0, CR1, CR5 and CR6 and reserves CR7 for use in yieldpoints. None of the compilers use CR2, CR3, or CR4 to avoid saving/restoring condition registers when doing a JNI transition from C to Java code.
 - **CR0, CR1, CR5, CR6, CR7** - volatile
 - **CR2, CR3, CR4** - non-volatile

Stack conventions

Stacks grow from high memory to low memory. The layout of the stackframe appears in a block comment in `ppc/VM_StackframeLayoutConstants.java`.

Calling Conventions

Parameters

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

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

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

Method prologue responsibilities

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

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

Method epilogue responsibilities

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

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

Linux/x86-32 VM Conventions

This section describes register, stack, and calling conventions that apply to Jikes RVM on Linux[®]/x86-32.

Register conventions

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

- **EDX:** Second GPR parameter register, second GPR result value (low-order part of a long result), otherwise volatile (caller-save).
- **EBX:** Nonvolatile.
- **ESP:** Stack pointer.
- **EBP:** Nonvolatile.
- **ESI:** Processor register, reference to the VM_Processor object for the current virtual processor.
- **EDI:** Nonvolatile. (used to hold JTOC in baseline compiled code)

Stack conventions

Stacks grow from high memory to low memory. The layout of the stackframe appears in a block comment in `ia32/VM_StackframeLayoutConstants.java`.

Calling Conventions

At the beginning of callee's prologue

The first two areas of the callee's stackframe (see above) have been established. ESP points to caller's return address. Parameters from caller to callee are as mandated by `ia32/VM_RegisterConstants.java`.

After callee's epilogue

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

OS X VM Conventions

Calling Conventions

The calling conventions we use for OS X are the same as those listed at:



<http://developer.apple.com/documentation/DeveloperTools/Conceptual/MachORuntime/MachORuntime.pdf>


They're similar to the Linux PowerPC calling conventions. One major difference is how the two operating systems handle the case of a long parameter when you only have a single parameter register left.

Magic

This page last changed on Mar 08, 2007 by [pdonald](#).

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 bytecode 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 `VM_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 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;

- [Raw Memory Access](#): Unfettered access to memory.
- [Uninterruptible Code](#): Declaring code to be uninterruptible.
- [Alternative Calling Conventions](#): Declaring different calling conventions and activation frame layout.

Compiler Intrinsics

This page last changed on Mar 13, 2007 by [pdonald](#).

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.

VM_Magic

All the methods in `VM_Magic` are compiler intrinsics. ecause these methods access raw memory or other machine state, perform unsafe casts, or are operating system calls, they cannot be implemented in Java code.

A Jikes™ RVM implementor must be *extremely careful* when writing code that uses `VM_Magic` to circumvent the Java type system. The use of `VM_Magic.objectAsAddress` to perform various forms of pointer arithmetic is especially hazardous, since it can result in pointers being "lost" during garbage collection. All such uses of magic must either occur in uninterruptible methods 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 `VM_Magic` methods are undefined. Instead, for each of these methods, the Java instructions to generate the code is stored in `OPT_GenerateMagic` and `OPT_GenerateMachineSpecificMagic` (to generate HIR) and `VM_Compiler` (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.

Raw Memory Access

This page last changed on Mar 07, 2007 by [pdonald](#).

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](#).

Unboxed Types

This page last changed on Mar 07, 2007 by [pdonald](#).

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

The Jikes RVM also defines a number of other 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.VM_Code`

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.
- 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.

Uninterruptible Code

This page last changed on Mar 07, 2007 by [pdonald](#).

What are the Semantics of Uninterruptible Code?

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

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

1. Because a stack overflow check represents a potential yield point (if GC is triggered when the stack is grown), stack overflow checks are omitted from the prologues of uninterruptible code. As a result, all uninterruptible code must be able to execute in the stack space available to them when the first uninterruptible method on the call stack is invoked. This is typically about 8K for uninterruptible regions called from mutator code. The collector threads must preallocate enough stack space, since all collector code is uninterruptible. As a result, using recursive methods in the GC subsystem is a bad idea.
2. Since no yield points are inserted in uninterruptible code, there will be no timer-driven thread switches while executing it. So, if possible, one should avoid "long running" uninterruptible methods outside of the GC subsystem.
3. Certain bytecodes are forbidden in uninterruptible code, because Jikes RVM cannot implement them in a manner that ensures uninterruptibility. The forbidden bytecodes are: *aastore* ; *invokeinterface* ; *new* ; *newarray* ; *anewarray* ; *athrow* ; *checkcast* and *instanceof* unless the LHS type is a final class ; *monitorenter* , *monitorexit* , *multianewarray* .
4. Uninterruptible code cannot cause class loading and thus must not contain unresolved *getstatic* , *putstatic* , *getfield* , *putfield* , *invokevirtual* , or *invokestatic* bytecodes.
5. Uninterruptible code cannot contain calls to interruptible code. As a consequence, it is illegal to override an uninterruptible virtual method with an interruptible method.
6. Uninterruptible methods cannot be synchronized.

We have augmented the baseline compiler to print a warning message when one of these restrictions is violated. If uninterruptible code were to raise a runtime exception such as `NullPointerException`, `ArrayIndexOutOfBoundsException`, or `ClassCastException`, then it could be interrupted. We assume that such conditions are a programming error and do not flag bytecodes that might result in one of these exceptions being raised as a violation of uninterruptibility. Checking for a particular method can be disabled by annotating the method with `org.vmmagic.pragmas.LogicallyUninterruptible`. This should be done with extreme care, but in a few cases is necessary to avoid spurious warning messages.

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

1. All class initializers are interruptible, since they can only be invoked during class loading.
2. All object constructors are interruptible, since they can only be invoked as part of the implementation of the `new` bytecode.
3. If a method is annotated with `org.vmmagic.pragmas.Interruptible` then it is interruptible.
4. If none of the above rules apply and a method is annotated with `org.vmmagic.pragmas.Uninterruptible`, then it is uninterruptible.

5. If none of the above rules apply and the declaring class is annotated with `org.vmmagic.pragmas.Uninterruptible` then it is uninterruptible.

Whether to annotate a class or a method with `org.vmmagic.pragmas.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 annotated the class is preferred.

This page last changed on Mar 11, 2007 by [pdonald](#).

The garbage collectors for the Jikes RVM are provided by MMTk. The [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.

The 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 RVM runtime.
- 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 semispaces, 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.

Virtual memory pages are lazily mapped into the 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. If the RVM scheduler is configured to run with N virtual processors (see [Thread Management](#)) capable of running N concurrent mutator threads then a `StopTheWorldGC` collection will be performed by N collector threads, each one running on its own virtual processor.

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.

The RVM does not currently support concurrent garbage collection.

The Jikes RVM may also use the [GCSpy](#) visualization framework. GCSpy allows developers to observe the behavior of the heap and related data structures.

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 JikesRVM 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:

- Create a GCspy *space*.
- Create a *stream* for each attribute of the space.
- Update the tile statistics as the memory manager passes it information.
- 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.

Installation of GCspy with Jikes RVM

System Requirements

The GCspy C server code needs a pthread (created in `gcspsyStartserver()` in `sys.C`) in order to run. So, GCspy will only work on a system where you've build Jikes RVM with `config.single.virtual.processor` set to 0. The build process will fail if you try to configure such a build.

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.gcspsy` must be set to 1 such as in the `BaseBaseSemiSpaceGCspyconfiguration`. 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.gcspsy-client` must be set to 1.

The following steps build the Jikes RVM with support for GCspy on linux-ia32 platform.

```
$ cd $RVM_ROOT
$ ant -Dhost.name=ia32-linux -Dconfig.name=BaseBaseSemiSpaceGCspy
-Dconfig.include.gcspsy-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.gcspsy-stub` to 1. 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 $RVM_ROOT/dist/BaseBaseSemiSpaceGCspy_ia32-linux
$ ./rvm -Xms20m -X:gc:gcspsyPort=3000 -X:gc:gcspsyWait=true &
```

Then you need to start the GCspy visualiser client.

```
$ cd $RVM_ROOT/dist/BaseBaseSemiSpaceGCspy_ia32-linux
$ ./tools/gcspsy/gcspsy
```

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.

Command line arguments

Additional GCspy-related arguments to the `rvm` command:

- `-X:gc:gcspsyPort=<port>`

The number of the port on which to connect to the visualiser. The default is port 0, which signifies

no connection.

- `-X:gc:gcspaceWait=<true/false>`

Whether Jikes RVM should wait for a visualiser to connect.

- `-X:gc:gcspaceTilesize=<size>`

How many KB are represented by one tile. The default value is 128.

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 needsLinearScan()` and `boolean withGCspy()`, both of which return `true`.

`SSGCspyTraceLocal` extends `SSTraceLocal` to override method `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.

1. Initialise the GCspy server: `server.init(name, portNumber, verbose);`
2. Add each event to the `ServerInterpreter` ('server' for short) `server.addEvent(eventID, eventName);`

3. Set some general information about the server (e.g. name of the collector, build, etc)
`server.setGeneralInfo(info);`
4. 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,

1. We may need to reset the GCspy driver's data depending on the collection phase.
2. 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`, `short` 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;

Care and Feeding

This page last changed on Mar 07, 2007 by [pdonald](#).

This section describes the practical aspects of getting started using and modifying the Jikes RVM. The [Quick Start Guide](#) gives a 10 second overview on how to get started while the following sections give more detailed instructions.

1. [Get The Source](#)
2. [Build the RVM](#)
3. [Run the RVM](#)
4. [Configure the RVM](#)
5. [Modify the RVM](#)
6. [Test the RVM](#)
7. [Debug the RVM](#)
8. [Evaluate the RVM](#)

Building the RVM

This page last changed on Mar 02, 2007 by [pdonald](#).

This guide describes how to build the 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 JikesRVM.

Overview

Compiling the source code

The majority of the 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 the Jikes RVM that is written in C that must be compiled with a C compiler such as gcc. The Jikes RVM uses [Ant](#) version 1.6.5 or later as the build tool that orchestrates the build process and executes the steps required in building the Jikes RVM.

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.

Bootstrapping the RVM

The Jikes RVM compiles Java class files and produces arrays of code and data. To build itself the Jikes RVM will execute on an existing Java Virtual Machine and compiles a copy of it's 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 the Jikes RVM.

Class libraries

The Java class library is the mechanism by which Java programs communicate with the outside world. The Jikes RVM uses the [GNU Classpath](#) class library. The developer can either specify a particular version of GNU Classpath to use or they can allow the build process to download and build an appropriate version of the library.

Target Requirements

The Jikes RVM is known to build and work on certain combinations of instruction architectures and operating systems. The following sections detail the supported architectures and operating systems.

Architectures

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

Intel

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 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.

The Jikes RVM currently supports the IA-32 ISA. As IA-32e is backward compatible with IA-32, the 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.

Operating Systems

The 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 AIX, Linux and OSX. At one stage a port to win32 was completed but it was never integrated into the main Jikes RVM codebase.

Support Matrix

The following table details the targets that have historically been supported and the current status of the support. The target.name column is the identifier that the Jikes RVM uses to identify this target.

target.name	Operating System	Instruction Architecture	Address Size	Status
ia32-linux	Linux	ia32	32 bits	OK
ia32-osx	OSX	ia32	32 bits	Unknown
ppc32-aix	AIX	PowerPC	32 bits	OK*
ppc32-linux	Linux	PowerPC	32 bits	OK
ppc32-osx	OSX	PowerPC	32 bits	Not Working
ppc64-aix	AIX	PowerPC	64 bits	OK*

ppc64-linux	Linux	PowerPC	64 bits	Not Working
x86_64-linux	Linux	ia32	64 bits	OK

* Building GNU classpath 0.93 on AIX requires minor edits to fdlibm.h and to files in native-lib. We're working on getting those changes merged back into GNU classpath so that future versions will work out-of-the-box. In the interim, if you need help figuring out how to build on AIX contact Dave Grove.

Tool Requirements

Java Virtual Machine

The Jikes RVM requires an existing Java Virtual Machine that conforms to Java 5.0 such as [Sun JDK 1.5](#) or IBM JDK 5.0. 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 this Java Virtual Machine on your path. You can run "java -version" to check you are using the correct JVM.

Ant

[Ant](#) version 1.6.5 or later is the tool required to orchestrate the build process. You can download and install the Ant tool from <http://ant.apache.org/> if it is not already installed on your system. The remaining build instructions assume that \$ANT_HOME/bin is on your path. You can run "ant -version" to check you are running the correct version of ant.

C Tool Chain

The Jikes RVM assumes that the GNU C Tool Chain is present on the system or a tool chain that is reasonably compatible. Most modern *nix environments satisfy this requirement.

Bison

As part of the build process the 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.

Instructions

Defining Ant properties

There are a number of ant properties that are used to control the build process of the 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. The following table describes some properties that are commonly specified.

Property	Description	Default
host.name	The name of the host environment used for building the Jikes RVM. The name should match one of the files located in the build/hosts/ directory minus the '.properties' extension.	None
target.name	The name of the target environment for the Jikes RVM. The name should match one of the files located in the build/targets/ directory minus the '.properties' extension. This should only be specified when cross compiling the Jikes RVM. See Cross-Platform Building for a detailed description of cross compilation.	\${host.name}
config.name	The name of the configuration used when building the Jikes RVM. The name should match one of the files located in the build/configs/ directory minus the '.properties' extension. This setting is further described in the section Configuring the 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 the Jikes RVM.	""
components.dir	The directory where Ant looks for external components when building the RVM.	\${jikesrvm.dir}/components
dist.dir	The directory where Ant stores	\${jikesrvm.dir}/dist

	the final Jikes RVM runtime.	
build.dir	The directory where Ant stores the intermediate artifacts generated when building the Jikes RVM.	\${jikesrvm.dir}/target
protect.config-files	Define this property if you do not want the build process to update configuration files when auto downloading components.	(Undefined)

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.

Selecting a Configuration

A "configuration" in terms of the Jikes RVM is the combination of build time parameters and component selection used for a particular Jikes RVM image. The [Configuring the 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 the 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=..."`.

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 manually 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.

Building the RVM

The next step in building the 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}". The following table describes some of the ant targets that can be executed. A complete list of documented targets can be listed by executing the command "ant -projecthelp"

Target	Description
check-properties	Check that all the required properties are defined.
compile-mmtk	Compile MMTk toolkit.
prepare-source	Generate configuration independent source code if required or force.generation property is set.
prepare-config-source	Generate source code for the current configuration if required or force.generation property is set.
main or runtime	Build a runtime image.
clean	Remove the intermediate directory and runtime image directory.
real-clean	Remove all generated artifacts.

Running the RVM

The 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 the RVM](#) for a complete list of command line options.

Building Patched Versions

This page last changed on Feb 24, 2007 by [pdonald](#).

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}_${patch.name}` rather than `${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 $RVM_ROOT
% ant -Dconfig.name=prototype -Dhost.name=ia32-linux
% patch -p0 < ReadBarriers.diff
% ant -Dpatch.name=ReadBarriers -Dconfig.name=prototype -Dhost.name=ia32-linux
% patch -R -p0 < ReadBarriers.diff
```

The `patch.name` property is also supported and reported as part of the test infrastructure.

Cross-Platform Building

This page last changed on Mar 08, 2007 by [pdonald](#).

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

Because 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 AIX™ on a Linux host:

```
% cd $RVM_ROOT
% 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 $RVM_ROOT
% 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.

Primordial Class List

This page last changed on Mar 02, 2007 by [pdonald](#).

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;

1. 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.
2. 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.
3. Enough of the core VM services and data structures, and class library (java.*) and 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, those classes that start with VMOpt that absolutely have to 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.

Configuring the RVM

This page last changed on Mar 08, 2007 by [pdonald](#).

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.

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 <i>opt</i> or <i>base</i> .	base
config.bootimage.compiler.args	Parameter specifies any extra args that are passed to the bootimage compiler.	""
config.runtime.compiler	Parameter selects the compiler used at runtime. Must be either <i>opt</i> or <i>base</i> .	base
config.include.aos	Include the adaptive system if set to 1. Parameter will be ignored if config.runtime.compiler is not <i>opt</i> .	0
config.assertions	Parameter specifies the level of assertions in the code base. Must be one of <i>extreme</i> , <i>normal</i> or <i>none</i> .	normal
config.include.all-classes	Include all the Jikes RVM classes in the bootimage if set to 1.	0
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.include.gcspy	Set to 1 to build RVM with GC Spy support. See Using GC Spy for more details.	0
config.include.gcspy-client	Set to 1 to bundle the GC Spy client with the Jikes RVM build. Parameter will be ignored if config.include.gcspy is not 1.	0
config.include.gcspy-stub	Set to 1 to use the GC Spy stub rather than the real GC Spy component. Parameter will be ignored if config.include.gcspy is	0

	not 1.	
config.stress-gc	Build will be a stress test for the GC subsystem if set to 1.	0
config.mmtk.plan	The name of the GC plan to use for the build. See MMTk for more details.	None

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.

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 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.

Configurations in Depth

Most standard Jikes RVM configuration files loosely follow the following naming scheme:

`<boot image compiler> Base|"Adaptive" <garbage collector>`
 where

- the `<boot image compiler>` is the compiler used to compile Jikes RVM's boot 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.

The following garbage collection suffixes are available:

- "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"**.

Some files augment the standard configurations as follows:

- The word **"Full"** at the beginning of the configuration name identifies a configuration such that all the Jikes RVM classes are included in the boot image. (By default only a small subset of these classes are included in the boot image.)
- **"FullAdaptive"** images have all of the included classes already compiled by the optimizing compiler.
- **"FullBaseAdaptive"** images have the included classes already compiled by the baseline compiler; the adaptive system will later recompile any hot methods.
- The word **"Fast"** at the beginning of the configuration name identifies a **"Full"** configuration where all assertion checking has been turned off. Note: **"Full"** and **"Fast"** boot images run faster but take longer to build.
- Prefixing the configuration with **"ExtremeAssertions"** indicate that the `config.assertions` configuration parameter is set to `extreme`. This turns on a number of expensive assertions.

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, 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=OptAdaptiveSemiSpace
```

Debugging the RVM

This page last changed on Mar 07, 2007 by [pdonald](#).

There are different tools for debugging the Jikes RVM:

GDB

There is a limited amount of C code used to start the Jikes RVM. The rvm script will start the 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 determine the location use the RVM.map file. A script to enable use of the RVM.map as debugger information inside GDB is provided [here](#).

Java Platform Debugger Architecture (JPDA)

In general the JPDA provides 3 mechanisms for debugging Java applications:

- The [Java Debug Interface](#) is an API for debugging Java code from Java.
- The [JVM Tools Interface](#) is an API for writing native/C code for debugging a JVM, it is similar to the Java Native Interface (JNI).
- The [Java Debug Wire Protocol](#) is a network protocol for debugging Java code running on JVMs.

Currently JDWP code is being implemented in the Jikes RVM based on the GNU Classpath implementation.

Experimental Guidelines

This page last changed on Mar 07, 2007 by [pdonald](#).

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

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.
- Any configuration that performs opt compilation at runtime (`config.include.aos=1` should be built with `config.include.all-classes=1`). This includes the optimizing compiler and associated support classes in the boot image where they can be optimized by the boot image compiler. By convention, configurations that include the opt compiler in the boot image have the Full or Fast prefix. Configurations where `config.include.all-classes` is not set to 1 that use the optimizing compiler will dynamically load it (which will force it to be baseline compiled).

What command-line arguments should I use?

For best performance we recommend the following:

- `-X:processors=all`: By default, Jikes™ RVM uses only one processor. Setting this option tells the runtime system to utilize all available processors.
- Set the heap size generously. We typically set the heap size to at least half the physical memory on a machine.
- Use a dedicated machine with no other users. The Jikes RVM thread and synchronization implementation do not play well with others.

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.

Here is how to use it:

1. Generate the profile information, using the following command line arguments:

For edge profile

```
-X:base:edge_counters=true  
-X:base:edge_counter_file=my_edge_counter_file
```

For adaptive compilation profile

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

For dynamic call graph profile (used by adaptive inlining)

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

2. Use the profile you generated for compiler replay, using the following command line arguments:

```
-X:aos:enable_replay_compile=true  
-X:vm:edgeCounterFile=my_edge_counter_file  
-X:aos:cafi=my_compiler_advice_file  
-X:aos:dcfi=my_dynamic_call_graph_file
```

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

Perhaps you are not seeing stellar Jikes™ RVM performance. If Jikes RVM as described above is not competitive with the IBM AIX™ or Linux®/x86-32 product DK, we recommend you test your installation with the SPECjvm™98 benchmarks. We expect Jikes RVM performance to be competitive with the IBM® DK 1.3.0 on the SPECjvm98 benchmarks.

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

The x86-32/IA-32 port is somewhat less mature than the PPC port, and does not deliver competitive performance on some codes. In particular, x86 floating-point performance is mediocre.

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

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

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

Get The Source

This page last changed on Feb 24, 2007 by [pdonald](#).

The source code for the Jikes RVM is stored in a [Subversion](#) repository. A developer can either work with the version control system or downloaded one of the releases.

Download a Release

Major and minor releases of the 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 the 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
```

Use Subversion

The source code for the Jikes RVM is stored in a [Subversion](#) repository. There is plenty of online documentation for using Subversion such as the [Subversion Book](#) or [Sourceforge documentation](#) but after installing Subversion the current version of source can be downloaded via:

```
> svn co https://svn.sourceforge.net/svnroot/jikesrvm/rvmroot/trunk jikesrvm
```

You can also retrieve a specific version (such as 2.4.6) via:

```
> svn co https://svn.sourceforge.net/svnroot/jikesrvm/rvmroot/tags/JIKESRVM-2r4r6 jikesrvm
```

If you are a core developer you will be able to commit changes to the repository directly. Otherwise you will need to place patches in the [patch tracker](#).

You can browse the online subversion repository at <http://svn.sourceforge.net/viewvc/jikesrvm/rvmroot/trunk/>.

Use SVK

You may want to retrieve Jikes RVM, make modifications and version control your changes. Unfortunately Subversion does not natively support decentralized development model where everyone maintains their

own development tree. For this you will need to use [SVK](#). SVK describes itself as "a decentralized version control system built with the robust Subversion filesystem. It supports repository mirroring, disconnected operation, history-sensitive merging, and integrates with other version control systems, as well as popular visual merge tools."

Note: The directions on this page were partially derived from a blog [entry](#) by Scott Laird.

Setting up the Local Repository

After you have downloaded and installed SVK you need to initialize a local repository and import Jikes RVM into local repository.

```
> svk depotmap --init # Initialize local repository
> svk mirror https://svn.sourceforge.net/svnroot/jikesrvm/rvmroot/trunk //jikesrvm/trunk # Setup up mapping
> svk sync --all # Synchronize local repository to all remote repositories
```

Synchronization will take a long time (Jikes RVM has a lot of history to import) and may need to be restarted if there is problems with the sourceforge servers.

Setting up a Local Branch

After the remote repository has been synchronized you can setup a local branch where you will do all your work and check out a working copy.

```
> svk cp //jikesrvm/trunk //jikesrvm/local # Setup local branch
> svk co //jikesrvm/local ~/Research/jikesrvm # Check out local branch into directory
~/Research/jikesrvm
```

SVK differs from Subversion in that the metadata for a checkout is not stored in the directory in which the files are checked out. You will not find any files such as `~/Research/jikesrvm/.svk/` - instead it is stored in central location. This means that if you ever want to move around a locally checked out copy you need to run:

```
> svk co --relocate ~/Research/jikesrvm ~/Research/jikesrvm2
```

And if you ever need to delete a locally checked out copy you delete the physical directory and then purge the record of its existence via:

```
> rm -rf ~/Research/jikesrvm2
> svk co --purge
```

Making Local Changes

You can modify the files in the checked out working copy and perform all the usual "svk commit", "svk diff" commands on the source code.

```
> cd ~/Research/jikesrvm
> vi NEWS
> svk diff NEWS
> svk commit -m "Added a news item" NEWS
```

Merging with Upstream Changes

When changes occur in the upstream Jikes RVM subversion repository you can update your local repository via the command:

```
> svk sync //jikesrvm/trunk
```

And you may want to move the changes across into your local branch via:

```
> svk smerge -I //jikesrvm/trunk //jikesrvm/local
```

And update your local branch via:

```
> cd ~/Research/jikesrvm
> svk up
```

Preparing Patches

Now that you have modified your local branch you want to create a patch so that it can be integrated back into the main Jikes RVM tree. The first thing to do is create a directory that will contain the patched versions.

```
> svk mkdir //jikesrvm/patchsets
```

Then you create a new branch for the patched version:

```
> svk cp //jikesrvm/trunk //jikesrvm/patchsets/news_add
```

And merge across the changes you are interested in:

```
> svk merge -l -c 1234 //jikesrvn/local //jikesrvn/patchsets/news_add
```

Note: the -l switch says to use the previous commit message as part of the new merge commit message.

You then check out the branch and verify your changes. By verifying your code against a clean tree you can make sure that you didn't miss any other changes in your local tree and that you didn't introduce any unintended changes. If you missed some changes you can add them directly to the branch or merge them across.

```
> svk co //jikesrvn/patchsets/news_add
> cd news_add
> ... RunSanityTests ...
> ... svk merge any missed changes ...
> ... emacs NEWS ...
> ... svk commit -m "Fix missed changes" ...
```

Finally you create a diff file and add it into the [patch tracker](#).

```
> svk diff //jikesrvn/trunk //jikesrvn/patchsets/news_add > news_add_diff.txt
```

If the main source moved further forward and you need to update your local branch you can do it via the following command. Then just retest and recreate the diff file.

```
> svk smerge -I -l //jikesrvn/trunk //jikesrvn/patchsets/news_add
```

Applying Changes

If you are a Jikes committer and want to apply patch directly to the upstream source repository then you run:

```
> svk smerge //jikesrvn/patchsets/news_add //jikesrvn/trunk
```

Mirroring Local Branch

Sometimes you may want to mirror a local branch into a public Subversion repository. This may be so others can look at your changes, to provide a backup or to share changes between multiple SVK repositories.

```
> svk mkdir //mirror
> svk ls http://www.mysvnserver.com/svn/public/
<follow the prompts and have it mirror it onto //mirror/mymirror>
> svk sync //mirror/mymirror
> svk mkdir //mirror/mymirror/jikesrvn
> svk smerge --baseless -I //jikesrvn/ //mirror/mymirror/jikesrvn
```

You can update the mirror at anytime via:

```
> svk smerge -I -I //jikesrvn/ //mirror/mymirror/jikesrvn
```

Use darcs

While a central SVN repository is excellent for collaboration on a common code base, sometimes you wish to try things before committing to the SVN repository. If you plan on making a lot of changes spread accross numerous files, it can be handy to have a local version control system (VCS) in which to record your changes. Similarly to the SVK setup detailed above, you can opt to use [darcs](#) as your local version control system. Darcs is an open-source distributed VCS. This means that there is no centralized server on which the repository resides. Instead, each repository stands on its own, and changes made to one copy of the repository can be put into another copy. Basically, darcs is built around patches, that are invertible, commutable (if not there is a dependency) and mergable in any order. Because people are pretty familiar wirth SVN, a there is a page detailing the [workflow comparison](#) between darcs and SVN.

The directions given here are structured alike the SVK directions.

Setting up a Local Repository

Once you have downloaded and installed darcs onto your machine, you can create the initial repository:

```
> cd my_jikes_source_dir
> darcs init
```

Make sure you have only the source files in your my_jikes_source_dir directory, as we will be automatically adding all source files to the darcs repository. Darcs ignores all CVS and SVN related files, as well as a host of other so called boring files. The list of boring files is kept in the prefs/boring file in your darcs repository. Feel free to expand this list.

```
> darcs record --look-for-adds
```

If you give no other parameters, you get the following questions


```
> What is your email address? <snip>
> addfile ./GNUmakefile
> Shall I record this patch? (1/?) [ynWsfqadjkc], or ? for help: a
> What is the patch name? initial checkin
> Do you want to add a long comment? [yn] n
```

You will only be asked for your email address the first time you check something into the repository. After the initial checkin, you end up with a `_darcs` directory that contains all the relevant information about your repository.

Making Local Changes

If you edit files locally, have tested your changes (i.e., Jikes RVM at least builds), you may want to record them in your repository. First check to see that you will record what you think has been changed.

```
> darcs whatsnew
```

If you added files in the Jikes RVM tree, you may wish to use the `--look-for-adds` argument to `whatsnew`. There are no special issues when dealing with binary files. If you are satisfied you can record the changes to all changes files (optionally using the `--look-for-adds` flag).

```
> darcs record
```

Or you can only record the changes made to one or more file, by explicitly adding them to the command line

```
> darcs record file1 file2 file3 ... fileN
```

You can confirm each change, or confirm them all at once, by typing a when asked to record 'this' patch. Notice that the smaller patches will be congregated into the larger patch that will get a name and (optionally) a comment in the repository. You can rollback patches, undoing them in the repository, the local copy (i.e., the working copy), or both. A list with patches you have recorded is available.

```
> darcs changes
```

Merging with Upstream Changes

If the Jikes RVM central SVN repository is changed, you use the regular SVN update to get the changes, and then you simply record the changes into the local darcs repository using

```
> darcs record -m "svn update at <date>" --look-for-adds
```

Preparing Patches and Applying Changes

If you wish to get a patch to the Jikes RVM team, you first update your local copy using SVN, store the changes into darcs, as described above, and basically create a SVN diff and submit the diff to the [patch tracker](#).

If you have SVN write access to the central Jikes RVM SVN repository, you use the regular SVN command to check in your code.

Mirroring a Local Branch

Your darcs repository can be made available though HTTP or SSH. For the former, you place the `_darcs` directory on a public HTTP server, for the latter you need to give people SSH access to the machine your repository resides on. If you wish to take a copy of your repository along, you basically have two choices. Either you copy the full Jikes RVM source tree, including your `_darcs` directory. This will allow you to keep track of the changes made in the central SVN repository while you are on the road. Or you only take your darcs repository along, relying on access to the machine on which the full Jikes RVM SVN checked out copy resides. For the latter you just 'get' your darcs repository. For the sake of clarity, we will call the machine you did all the above on the 'original_machine', and we will call the one where you want a copy on the 'road_machine'. On road_machine you

```
> darcs get original_machine:my_jikes_source_dir
```

If you make changes while on the road, you record them locally, on road_machine. If you wish to sync with original_machine you

```
> darcs put original_machine:my_jikes_source_dir
```

Suppose you have updated the working copy from the SVN repository on original_machine, and you wish these changes reflected on your road_machine, you first check the changes into darcs on the original_machine and then you

```
> darcs pull original_machine:my_jikes_source_dir
```

The darcs repositories with which you have synched are kept in `_darcs/prefs/repos`.

Use SVN::Mirror to Clone the SVN Repository

You can use `SVN::Mirror` to mirror the repository locally. You can install it via your distributions package

manager or via Perl's CPAN (`<code>perl -MCPAN -e shell</code>`) installation tool. You will need the command line interface to this called [svn](#).

First create a repository to hold your mirror that will be held at the location 'SVMREPOS':

```
> export SVMREPOS=~/.Research/jikesrvm
> svnadmin create $SVMREPOS
```

then initialize and synchronize it to the latest repository:

```
> svn init mirror/mymirror/jikesrvm https://svn.sourceforge.net/svnroot/jikesrvm/rvmroot/trunk
> svn sync mirror/mymirror/jikesrvm
```

Synchronization will take a long time (Jikes RVM has a lot of history to import) and may need to be restarted if there is problems with the sourceforge servers. You can speed up the sync by flattening a series of changes into one larger change via:

```
> svn sync mirror/mymirror/jikesrvm $REVISION
```

where \$REVISION is the current revision.

Rsync the SVN repository

The following commands will copy the Jikes RVM SVN repository to the current directory:

```
> export RSYNC_PROXY=rsync-svn.sourceforge.net:80
> rsync -a rsync-svn-j::svn/jikesrvm/* .
```

Modifying the RVM

This page last changed on Mar 07, 2007 by [pdonald](#).

The following sections give a rough overview on existing coding conventions.

- [Coding Style](#)
- [Coding Conventions](#)



Warning

Jikes RVM is a bleeding-edge research project. You will find that some of the code does not live up to product quality standards. Don't hesitate to help rectify this by contributing clean-ups, bug fixes, and missing documentation to the project. We are in the process of consolidating and simplifying the codebase at the moment.

Coding Conventions

This page last changed on Mar 07, 2007 by [pdonald](#).

The VM_prefix

By convention, any class which should be loaded into the boot image starts its name with `VM_`. At the time of writing this is used by the build process to determine the set of [classes](#) that should be written into Jikes RVM's boot image. The MMTk classes don't follow this naming convention because they're an independent subsystem, but the rest of Jikes RVM does.

Assertions

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) VM._assert(false, message);
```

An assertion failure is always followed by a stack dump.

Coding Style

This page last changed on Mar 07, 2007 by [pdonald](#).

Regrettably, much code in the current system does not follow any consistent coding style. This is an unfortunate residuum of the system's evolution. It makes editing sometimes unpleasant, and prevents Javadoc from formatting comments in many files. To alleviate this problem, we present this style guide for new Java™ code; it's just a small tweak of Sun®'s style guide.

File Headers

Every file needs to have a header with a copyright notice and one or more `@author` tags. There may additionally be a `@modified` tag for someone who modified code but doesn't want to claim co-authorship. (`@modified` is our own extended tag.)

A Java example of the notices follows.

```
/*
 *
 * This file is part of Jikes RVM (http://jikesrvm.sourceforge.net).
 * The Jikes RVM project is distributed under the Common Public License (CPL).
 * A copy of the license is included in the distribution, and is also
 * available at http://www.opensource.org/licenses/cpl1.0.php
 *
 * (C) Copyright IBM Corp 2003
 */
package org.jikesrvm;

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

/**
 * TODO Substitute a brief description of what this program or library does.
 *
 * @author Your Name Here
 */
```

Coding style description

The Jikes™ RVM coding style guidelines are defined with reference 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:

1. **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 is to be no tabs in the source files.
2. **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.
3. **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. There's an example elsewhere in this document.
4. **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 `VM_Configuration` class and the `VM_Properties` interface also are in that format. Since the `VM` class inherits fields from both `VM_Properties` and `VM_Configuration`, that's how we get `VM.VerifyAssertions`, etc.

Javadoc requirements

All 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. All classes and methods should have a block comment describing them
2. All methods contain a short description of their arguments (using `@param`), the return value (using `@return`) and the exceptions they may throw (using `@throws`).
3. Each class should include `@see` and `@link` references as appropriate.

Profiling Applications with Jikes RVM

This page last changed on Mar 11, 2007 by [pdonald](#).

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.

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 `or.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
```

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.

1. **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`.

1. **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`.

1. **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.

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

For an example, see `OPT_Inliner.java`. Look for the code guarded by the flag `COUNT_FAILED_METHOD_GUARDS`. Enabled by `-X:aos:insert_debugging_counters=true`.

Writing new instrumentation phases

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

Counter Managers:

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

Managed Data:

The class `VM_ManagedCounterData` is used to keep track of counter data that is managed using a counter manager. This purpose of the data object is to maintain the mapping between the counters themselves (which are indexed by number) and the events that they represent. For example, `VM_StringEventCounterData` is used record the fact that counter #1 maps to the event named "FooBar". Depending on what you are counting, there may be one data object for the whole program (such as `VM_YieldpointCounterData` and `VM_MethodInvocationCounterData`), or one per method. There is also a generic data object called `VM_StringEventCounterData` that allows events to be give string names (see Debugging Counters above).

Instrumentation Phases:

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

Lower Instrumentation Phase:

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

Quick Start Guide

This page last changed on Mar 07, 2007 by [pdonald](#).

1. `svn co https://jikesrvm.svn.sourceforge.net/svnroot/jikesrvm/rvmroot/trunk jikesrvm`
2. `cd jikesrvm`
3. `echo "host.name=ia32-linux" > .ant.properties` # Change this to match appropriate host
4. `ant main -Dconfig.name=prototype-opt` # Change this to select appropriate configuration
5. `./dist/prototype-opt_ia32-linux/rvm -version` # Change dir to use selected host and configuration

Running the RVM

This page last changed on Mar 08, 2007 by [pdonald](#).

Jikes™ RVM executes Java virtual machine byte code instructions from `.class` files. It does *not* compile Java™ source code. Therefore, you must compile all Java source files into bytecode using your favorite 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:`.

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.

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>

Non-Standard Command-Line Options

The non standard command-line options are grouped according to the subsystem that they control. The following sections list the available options in each group.

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 com.ibm.jikesrvm.* and java.* classes from <path>
-X:cpuAffinity=<number>	The physical CPU to which first virtual processor is bound
-X:processors=<number "all">	The number of virtual processors

Memory Non-Standard Command-Line Options

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

Garbage Collector Non-Standard Command-Line Options

Option	Description
--------	-------------

Base Compiler Non-Standard Command-Line Options

Option	Description
--------	-------------

Opt Compiler Non-Standard Command-Line Options

Option	Description
--------	-------------

Adaptive System Non-Standard Command-Line Options

Option	Description
--------	-------------

Testing the RVM

This page last changed on Mar 07, 2007 by [pdonald](#).

The Jikes RVM includes a testing framework for running functional and performance tests and it also includes a number of actual tests. See [External Test Resources](#) for details or downloading prerequisites for the 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 the 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 the RVM](#) for details) combined with a set of parameters that are passed to the 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:opt:O1` 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 the 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.

Ant Properties

There is a number of ant properties that control the test process. Besides the properties that are already defined in [Building the RVM](#) the following properties may also be specified.

Property	Description	Default
test-run.name	The name of the <i>test-run</i> . The name should match one of the files located in the build/test-runs/ directory minus the '.properties' extension.	sanity
results.dir	The directory where Ant stores the results of the test run.	<code>\${jikesrvm.dir}/results</code>
results.archive	The directory where Ant gzips and archives a copy of test run results and reports.	<code>\${results.dir}/archive</code>
send.reports	Define this property to send reports via email.	(Undefined)
mail.from	The from address used when emailing report.	<code>jikesrvm-core@lists.sourceforge.net</code>
mail.to	The to address used when emailing report.	<code>jikesrvm-regression@lists.sourceforge.net</code>
mail.host	The host to connect to when	localhost

	sending mail.	
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)

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>*. and the following table defines the possible properties.

Property	Description	Default
tests	The names of the test groups to execute.	None
name	The unique identifier for <i>test-configuration</i> .	""
configuration	The name of the RVM build configuration to test.	<tag>
mode	The test mode. May modify the way test groups execute. See individual groups for details.	""
extra.args	Extra arguments that are passed to the RVM.	""



Note

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}
```

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 a property must be specified either in .ant.properties or in the test-run properties file. The following patterns may be used to exclude particular tests.

Property	Description
exclude.test.<group>.<test>	Exclude single test.
exclude.test.<build-configuration>.<group>.<test>	Exclude single test in particular build configuration.
exclude.test.<build-target>.<group>.<test>	Exclude single test on particular target platform.
exclude.test.<build-configuration>.<build-target>.<group>.<test>	Exclude single test in particular build configuration on particular target platform.
exclude.group.<group>	Exclude test group.
exclude.group.<build-configuration>.<group>	Exclude test group in particular build configuration.
exclude.group.<build-target>.<group>	Exclude test group on particular target platform.
exclude.group.<build-configuration>.<build-target>.<group>	Exclude test group in particular build configuration on particular target platform.
exclude.<build-configuration>	Exclude all tests in particular build configuration.
exclude.<build-target>	Exclude all tests on particular target platform.
exclude.<build-configuration>.<build-target>	Exclude all tests in particular build configuration on particular target platform.

At the time of writing the Jikes RVM does not support suspending and resuming threads and as a result the test named "TestSuspend" 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 "exclude.test.basic.TestSuspend=true" into test-run properties file.

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 `target/test-driver/results/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.

External Test Resources

This page last changed on Feb 24, 2007 by [pdonald](#).

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.

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."

Note: There is a page that tracks how JikesRVM is doing in Dacapo
<http://cs.anu.edu.au/people/Robin.Garner/dacapo/regression/>

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://optusnet.dl.sourceforge.net/sourceforge/dacapobench/dacapo-2006-10.jar
```

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

xerces

Process some large documents using xerces XML parser.

```
> cd $BENCHMARK_ROOT
> 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 $BENCHMARK_ROOT/xmlFiles
> cd $BENCHMARK_ROOT/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 "Soot is a Java bytecode analysis and transformation framework. It provides a Java API for building intermediate representations (IRs), analyses and transformations; also it supports class file annotation."

```
> mkdir -p $BENCHMARK_ROOT/soot-2.2.3
> cd $BENCHMARK_ROOT/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 $BENCHMARK_ROOT/JavaGrandeForum
> cd $BENCHMARK_ROOT/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 $BENCHMARK_ROOT/JavaGrandeForum
> cd $BENCHMARK_ROOT/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 $BENCHMARK_ROOT/JLex-1.2.6/classes/JLex
> cd $BENCHMARK_ROOT/JLex-1.2.6/classes/JLex
> wget http://www.cs.princeton.edu/~appel/modern/java/JLex/Archive/1.2.6/Main.java
> mkdir -p $BENCHMARK_ROOT/QBJC
> cd $BENCHMARK_ROOT/QBJC
> wget http://www.ocf.berkeley.edu/~horie/qbjlex.txt
> mv qbjlex.txt qb1.lex
```

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;

```
$RVM_ROOT/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 $BENCHMARK_ROOT/SPECjbb2005  
> cd $BENCHMARK_ROOT/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???
```