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
      - AOS Controller
      - Cost Benefit Model
      - Jikes RVM's compilers
      - Life Cycle of a Compiled Method
      - Logging and Debugging
      - Threading and Yieldpoints
    - Compilers
      - Baseline Compiler
      - JNI Compiler
      - Optimizing Compiler
        - BURS
        - Compiler Optimization Comparison Chart
        - IR
        - Method Compilation
        - 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

User Guide

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 these Jikes RVM papers, focusing on implementation details of how to build, run, and add functionality to the system.

You may find sections of the User Guide missing, incomplete or otherwise confusing. We intend this document to live as a continual work-in-progress, hopefully growing and maturing as community members edit and add to the guide. Please accept this invitation to contribute.

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

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

Architecture

This page last changed on Mar 24, 2008 by steveblackburn.

This section describes the architecture of 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 Jul 07, 2008 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. This section of the userguide is based on section 5 of the 2004 technical report.

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

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

Selective Optimization

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

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

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

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

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

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

dynamically adjusted this threshold value and the size of the sampling window in an attempt to reduce the overhead of processing the samples.

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

Profile-Directed Inlining

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

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

The system currently used is based on Arnold & Grove's CGO 2005 paper. More details of the sampling scheme and the inlining oracle can be found there, or in the source code.

AOS Controller

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

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

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

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

Cost Benefit Model

The Jikes RVM Adaptive Optimization System attempts to evaluate the break-even point for each action using an online competitive algorithm. It relies on an analytic model to estimate the costs and benefits of each selective recompilation action, and evaluates the best actions according to the model predictions online.

A key advantage of this approach is that it allows a designer to extend the simple "break-even" cost-benefit model to account for more sophisticated adaptive policies, such as selective compilation with multiple optimization levels, on-stack-replacement, and long-running analyses.

In general, each potential action will incur some *cost* and may confer some _benefit. For example, recompiling a method will certainly consume some CPU cycles, but could speed up the program execution by generating better code. In this discussion we focus on costs and benefits defined in terms of time (CPU cycles). However, in general, the controller could consider other measures of cost and benefit, such as memory footprint, garbage allocated, or locality disrupted.

The controller will take some action when it estimates the benefit to exceed the cost. More precisely, when the controller wakes at time t, it considers a set of n available actions, the set A = {A1, A2, ..., An}. For any subset S in P(A), the controller can estimate the cost C(S) and benefit B(S) of performing all actions Ai in S. The controller will attempt to choose the subset S that maximizes B(S) - C(S). Obviously S = {} has B(S) = C(S) = 0; the controller takes no action if it cannot find a profitable course.

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

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

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

The controller must choose an action for each m in M. The set of available actions is Actions = {Ajm | 0 <= j <= k, m in M}.

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

For S in Actions, define C(S) = Sum({s in S}C(s)). Given S, for each m in M, define A_min_m to be the action Ajm in S that minimizes T(Ajm). Then define T(S) = Sum({m in M}T(A_{min}_m).

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

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

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

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

Let M=(m1, ..., mk) be the k compiled methods. When the controller wakes at time t, each compiled method m has been sampled Sum(m) times. Let delta be the sampling interval, measured in seconds. The controller estimates that method m has executed delta Sum(m) seconds so far, and will execute for another delta Sum(m) seconds in the future.

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

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

Jikes RVM's compilers

Jikes RVM invokes a compiler for one of three reasons. First, when the executing code reaches an unresolved reference, causing a new
class to be loaded, the class loader invokes a compiler to compile the class initializer (if one exists). Second, the system compiles each method the first time it is invoked. In these first two scenarios, the initiating application thread stalls until compilation completes.

In the third scenario, the adaptive optimization system can invoke a compiler when profiling data suggests that *recompiling* a method with additional optimizations may be beneficial. The system supports both background and foreground recompilation. With background recompilation (the default), a dedicated thread asynchronously performs all recompilations. With foreground configuration, the system invalidates a compiled method, thus, forcing recompilation at the desired optimization level at the next invocation (stalling the invoking thread until compilation completes).

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

1. The goal of the *baseline* compiler is to generate code quickly. Thus, it translates bytecodes directly into native code by simulating Java's operand stack. It does not build an intermediate representation nor perform register allocation, resulting in native code that executes only somewhat faster than bytecode interpretation. However, it does achieve its goal of producing this code quickly, which significantly reduces the initial overhead associated with dynamic compilation.
2. The *optimizing* compiler translates bytecodes into an intermediate representation, upon which it performs a variety of optimizations. All optimization levels include linear scan register allocation and BURS-based instruction selection. The compiler's optimizations are grouped into several levels:
   - **Level 0** consists of a set of flow-sensitive optimizations performed on-the-fly during the translation from bytecodes to the intermediate representation and some additional optimizations that are either highly effective or have negligible compilation costs. The compiler performs the following optimizations during IR generation: constant, type, non-null, and copy propagation, constant folding and arithmetic simplification, branch optimizations, field analysis, unreachable code elimination, inlining of trivial methods (A trivial method is one whose body is estimated to take less code space than 2 times the size of a calling sequence and that can be inlined without an explicit guard.), elimination of redundant nullchecks, checkcasts, and array store checks. As these optimizations reduce the size of the generated IR, performing them tends to reduce overall compilation time. Level 0 includes a number of cheap local (The scope of a local optimization is one extended basic block.} optimizations such as local redundancy elimination (common subexpression elimination, loads, and exception checks), copy propagation, constant propagation and folding. Level 0 also includes simple control flow optimizations such as static basic block splitting, peephole branch optimization, and tail recursion elimination. Finally, Level 0 performs simple code reordering, scalar replacement of aggregates and short arrays, and one pass of intraprocedural flow-in-sens-i-tive copy propagation, constant propagation, and dead assignment elimination.
   - **Level 1** resembles Level 0, but significantly increases the aggressiveness of inlining heuristics. The compiler performs both unguarded inlining of final and static methods and (speculative) guarded inlining of non-final virtual and interface methods. Speculative inlining is driven both by class hierarchy analysis and online profile data gathered by the adaptive system. In addition, the compiler exploits ``preexistence'' to safely perform unguarded inlining of some invocations of non-final virtual methods *without* requiring stack frame rewriting on invalidation. It also runs multiple passes of some of the Level 0 optimizations and uses a more sophisticated code reordering algorithm due to Pettis and Hansen.
   - **Level 2** augments level 1 with loop optimizations such as normalization and unrolling; scalar SSA-based flow-sensitive optimizations based on dataflow, global value numbering, global common subexpression elimination, redundant and conditional branch elimination; and heap array SSA-based optimizations, such as load/store elimination, and global code placement. **NOTE: many of the O2 optimizations are disabled by default by defining them as O3 optimizations because they are believed to be somewhat buggy.**

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

Life Cycle of a Compiled Method

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

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

Each time a class is loaded, the system checks the invalidation database to identify the set of compiled methods to mark as obsolete,
because this classloading action invalidates speculative optimizations previously applied to that method.
 A method may transition from either *compiling* or *installed* to *obsolete* due to a classloading-induced invalidation. A method can also transition from *installed* to *obsolete* when the adaptive system selects a method for optimizing recompilation and a new compiled method is installed to replace it.

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

Logging and Debugging

This page last changed on Jul 07, 2008 by dgrove.

Complex non-deterministic systems such as the Jikes RVM adaptive system present challenges for system understanding and debugging. Virtually all of the profiling data collected by the runtime measurements component results from non-deterministic timer-based sampling at taken yieldpoints. The exact timing of these interrupts, and thus, the profile data that drives recompilation decisions, differs somewhat each time an application executes. Furthermore, many of the optimizations in the optimizing compiler rely on online profiles of conditional branch probabilities, i.e., the probabilities at the point in an execution when the recompilation occurs. Thus, because recompilations can occur at different times during each execution, a method compiled at the same optimization level could be compiled slightly differently on different runs.

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

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

```
30:..7047728888 Compiled read with baseline compiler in 0.20 ms
90:..7136817287 Controller notified that read(14402) has 4.0 samples
92:..7139813016  Doing nothing cost (leaving at baseline) to read is 40.0
92:..7139830219  Compiling read cost at O0=40.42, future time=49.81
92:..7139842466  Compiling read cost at O1=65.99, future time=72.58
92:..7139854029  Compiling read cost at O2=207.44, future time=213.49
110:..7166901172 Controller notified that read(14402) has 9.0 samples
111:..7168378722  Doing nothing cost (leaving at baseline) to read=90.0
111:..7168396493  Compiling read cost at O0=40.42, future time=61.54
111:..7168409562  Compiling read cost at O1=65.99, future time=80.81
111:..7168421097  Compiling read cost at O2=207.44, future time=221.06
111:..7168435937 Scheduling level 0 recompilation of read (priority=28.46)
112:..7169879779 Recompiling (at level 0) read
114:..7173293360  Recompiled (at level 0) read
150:..7227058078 Controller notified that read(14612) has 5.11 samples
151:..7228691160  Doing nothing cost (leaving at O0) to read=51.12
151:..7228705466  Compiling read cost at O1=66.26, future time=102.14
151:..7228717124  Compiling read cost at O2=208.29, future time=241.24


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


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

This sample shows an abbreviated subset of the log entries associated with the method read of the class spec.benchmarks._213_javac.ScannerInputStream, one of the hotter methods of the SPECjvm98 benchmark _213_javac. The first pair of numbers are the controller clock (number of timer

interrupts since execution began) and the value of the hardware cycle counter (Time.cycles()) for the log entry. These log entries show the cost-benefit values computed by the controller for various possible optimization actions and the progression of the method from baseline compilation through two recompilations (level 0 and then at level 1). For example, at time 92, we see four entries that give the estimated total future time (the sum of the compilation cost and the total future execution time in a method) for performing no recompilation and for each optimization level. Because the total future time for not recompiling (40) is less than the other alternatives (49.81, 72.58, and 213.49), the method is not scheduled for recompilation. However, at time 110, the method has been sampled more often. Thus, the total future time estimate is updated, resulting in two recompilation actions (level 0 and level 1) that are more attractive than taking no recompilation action. Because level 0 gives the least future time, this decision is chosen by placing a recompilation event in the recompilation priority queue. The priority for the event is the expected improvement of performing this recompilation, i.e., the difference between the future time for the new level and the future time for current execution (90 - 61.54 = 28.46).

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

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

Threading and Yieldpoints

This page last changed on Jul 07, 2008 by dgrove.

For each physical processor on the system, the system creates a pthread. Each pthread is associated with a virtual processor object that executes one or more Java threads in a *quasi-preemptive* manner, as follows. Each compiler generates *yield points*, which are program points where the running thread checks a dedicated bit in the virtual processor object to determine if it should yield to another thread. The compilers insert yield points in method prologues, method epilogues, and on loop backedges. Currently, the system sets the thread-switch bit approximately every 10ms.

The adaptive optimization system piggybacks on this yieldpoint mechanism to gather profile data. The thread scheduler provides an
extension point by which the runtime measurments component can install listeners that execute each time a yieldpoint is taken. Such listeners primarily serve to sample program execution to identify frequently-executed methods and call edges. Because these samples occur at well-known locations (prologues, epilogues, and loop backedges), the listener can easily attribute each sample to the appropriate
Java source method.

The Jikes RVM implementation introduces a weakness with this mechanism, in that samples can only occur in regions of code that have yieldpoints. Some low-level Jikes RVM subsystems, such as the thread scheduler and the garbage collector, elide yieldpoints because
those regions of code rely on delicate state invariants that preclude thread switching. These uninterruptible regions can distort sampling accuracy by artificially inflating the probability of sampling the first yieldpoint executed after the program leaves an uninterruptible region of code.

Compilers

This page last changed on Mar 09, 2007 by pdonald.

- Baseline Compiler
- JNI Compiler
- Optimizing Compiler

Baseline Compiler

General Architecture

The goal of the baseline compiler is to efficiently generate code that is "obviously correct." It also needs to be easy to port to a new platform and self contained (the entire baseline compiler must be included in all Jikes RVM boot images to support dynamically loading other compilers).
Roughly two thirds of the baseline compiler is machine-independent. The main file is `BaselineCompiler` and its parent `TemplateCompilerFramework`. The main platform-dependent file is `BaselineCompilerImpl`.

Baseline compilation consists of two main steps: GC map computation (discussed below) and code generation. 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. `TemplateCompilerFramework` contains the main code generation switch statement that invokes the appropriate *emit<bytecode>_* method of `BaselineCompilerImpl`.

GC Maps

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

Command-Line Options

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

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 Jul 02, 2008 by dgrove.

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 fileslocated 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:

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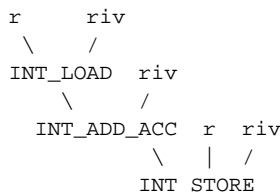
where `r` is a non-terminal that represents a register or a tree producing a register, `riv` is a non-terminal that represents a register (or a tree producing one) or an immediate value, and `INT_LOAD`, `INT_ADD_ACC` and `INT_STORE` are operators (*terminals*). `OTHER_OPERAND` is just an abstraction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

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

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

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

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

For more information on helper functions look at `BURS_Helpers.java`. For more information on the BURS algorithm see `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 25, 2008 by ianrogers.

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	✓ (disabled)	✓
	Loop striding	✗	✓
	Loop unrolling	✓ (disabled)	✓
	Loop peeling	✗	✓
	Loop versioning	✓ (disabled)	✓
	Loop specialization	✗	✓
	Exception directed optimization	✗	✓
	Switch analysis	✓	✓
Global optimizations	Global flow analyses and optimization	(?)	✓
	Partial redundancy elimination	✓ (disabled)	✓
	Escape analysis	✓	✓
	GC and memory allocation optimizations	✗	✓
	Synchronization optimizations	✓	✓
Native code generation	Small optimization based on architecture characteristics	(?)	✓

IR

This page last changed on Jul 02, 2008 by dgrove.

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

This section documents basic information about the intermediate 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 `Operators`, which in turn is automatically generated from a template by a driver. The input to the driver are two files, both called `OperatorList.dat`. One input file resides in `$RVM_ROOT/rvm/src-generated/opt-ir` and defines machine-independent operators. The other resides in `$RVM_ROOT/rvm/src-generated/opt-ir/${arch}` and defines machine-dependent operators, where `${arch}` is the specific instruction architecture of interest.

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

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

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

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

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

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

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

Instruction 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 `Operators` interface and the `Move` instruction format:

```
import com.ibm.jikesrvm.opt.ir.*;
class X {
```

```
  void foo(Instruction s) {
    if (Move.conforms(s)) {      // if this instruction fits the Move format
      RegisterOperand r1 = Move.getResult(s);
      Operand r2 = Move.getVal(s);
      System.out.println("Found a move instruction: " + r1 + " := " + r2);
    } else {
      System.out.println(s + " is not a MOVE");
    }
  }
}
```

This example shows just a subset of the access functions defined for the Move format. Other static access functions can set each operand (in this case, `Result` and `Val`), query each operand for nullness, clear operands, create Move instructions, mutate other instructions into Move instructions, and check the index of a particular operand field in the instruction. See the 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 `Operand`)
    - `[opt]`: is this operand optional?
- `VARSIG`: a description of repeating operands, used for variable-length instructions.

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

```
Move
1 0 1
"D Result RegisterOperand" "U Val Operand"
<blank line>
```

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

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

```
Call
1 0 3 1 U 4
"D Result RegisterOperand" \
"U Address Operand" "U Method MethodOperand" "U Guard Operand opt"
"Param Operand"
```

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

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

- `ResultCarrier`: provides access to an operand of type `RegisterOperand` named `Result`.
- `GuardResultCarrier`: provides access to an operand of type `RegisterOperand` named `GuardResult`.
- `LocationCarrier`: provides access to an operand of type `LocationOperand` named `Location`.
- `GuardCarrier`: provides access to an operand of type `Operand` named `Guard`.

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

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

Method Compilation

This page last changed on Jul 02, 2008 by dgrove.

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

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

The `OptimizationPlanner` class defines the standard phases used in a compilation. This class contains a static field, called `masterPlan`, which contains all possible `OptimizationPlanElements`. The structure of the master plan is a tree. Any element may either be an atomic element (a leaf of the tree), or an aggregate element (an internal node of the tree). The master plan has the following general structure:

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

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

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

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

Every optimization plan consists of a selection of elements from the master plan; thus two optimization plans associated with different methods will share the same component element objects. Clearly, it is undesirable to share state associated with a particular compilation phase between two different method compilations. In order to prevent this, the `perform` method of an atomic element creates a new instance of its phase immediately before calling the phase's `perform` method. In the case where the phase contains no state the `newExecution` method of `CompilerPhase` can be overridden to return the phase itself rather than a clone of the phase

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

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

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

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 `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 the 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 `IdleThread` and garbage collector thread `CollectorThread` into the idle queue and collector queue.

Next, the boot routine boots the 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

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 RVMClass object for this class has been created and registered and is in the process of being loaded.
- **loaded:** The class's bytecode file has been read and parsed successfully. The modifiers and attributes for the fields have been loaded and the constant pool has been constructed. The class's superclass (if any) and superinterfaces have been loaded as well.
- **resolved:** The superclass and superinterfaces of this class has been resolved. The offsets (whether in the object itself, the JTOC, or the class's TIB) of its fields and methods have been calculated.
- **instantiated:** The superclass has been instantiated and pointers to the compiled methods or lazy compilation stubs have been inserted into the JTOC (for static methods) and the TIB (for virtual methods).
- **initializing:** The superclass has been initialized and the class initializer is being run.
- **initialized:** The superclass has been initialized and the class initializer has been run.

Code Management

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

The JTOC

The JTOC holds pointers to each of Jikes™ RVM's global data structures, as well as literals, numeric constants and references to String constants. The JTOC also contains references to the TIB for each class in the system. Since these structures can have many types and the JTOC is declared to be an array of ints, Jikes RVM uses a descriptor array, co-indexed with the JTOC, to identify the entries containing references. The JTOC is depicted in the figure below.
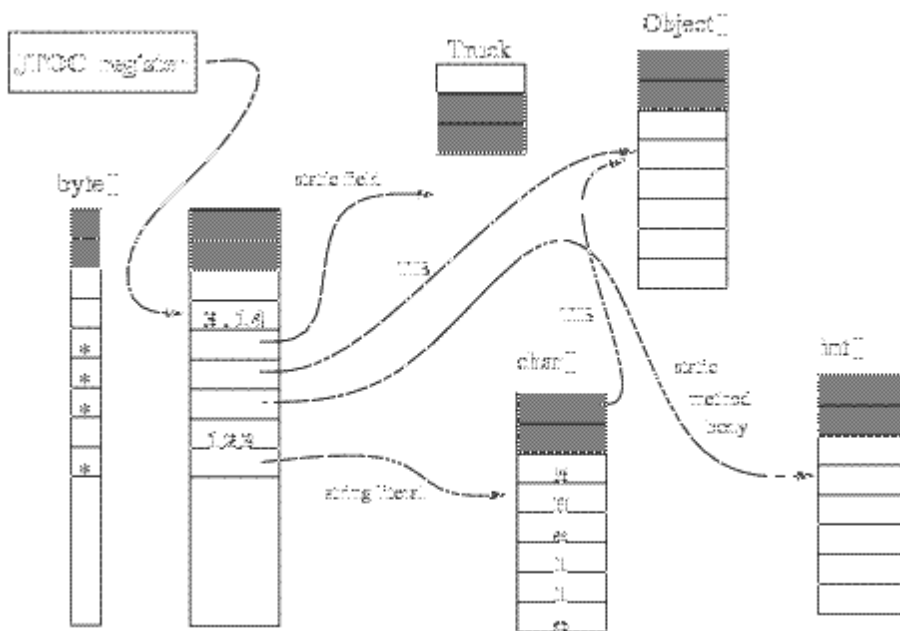
Virtual Methods

A TIB contains pointers to the compiled method bodies (executable code) for the virtual methods and other instance methods of its class. Thus, the TIB serves as Jikes RVM's virtual method table. A virtual method dispatch entails loading the TIB pointer from the object reference, loading the address of the method body at a given offset off the TIB pointer, and making an indirect branch and link to it. A virtual method is dispatched to with the *invokevirtual* bytecode; other instance methods are invoked by the *invokespecial* bytecode.

Static Fields and Methods

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

Instance Initialization Methods

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

Lazy Method Compilation

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

Interface Methods

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

The simplest, and least efficient way, of locating an interface method is to search all the virtual method entries in the TIB finding a match. Instead, Jikes RVM uses an *Interface Method Table*(IMT) which resembles a virtual method table for interface methods. Any method that could be an interface method has a fixed offset into the IMT just as with the TIB. However, unlike in the TIB, two different methods may share the same offset into the IMT. In this case, a *conflict resolution stub* is inserted in the IMT.

Conflict resolution stubs are custom-generated machine code sequences that test the value of a hidden parameter to dispatch to the desired interface method. For more details, see `InterfaceInvocation`.

Exception Management

This page last changed on Jul 02, 2008 by dgrove.

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 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 generationg for non-excepting 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 excepting 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 RuntimeEntrypoints.deliverHardwareException(). This method searches the stack from the excepting 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 RuntimeEntrypoints.athrow() so the unwind can never happens 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.

This page last changed on Jul 02, 2008 by dgrove.

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 `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 `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 `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 SysCall.java. Note again that the syscall methods are NOT JNI methods, but an independent (more efficient) interface that is specific to Jikes RVM.

The second mechanism is JNI. Naturally, the user writes JNI code using the JNI interface. RVM implements a call to a native method by using the platform-specific `JNICompiler` to generate a stub routine that manages the transition between Java bytecode and native code. A JNI call is a GC-safe point, since JNI code cannot freely modify the Java heap.

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

Things JNI Can't Handle

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

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

This page last changed on Jul 02, 2008 by dgrove.

Object Model

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

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 `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: `JavaHeader`, which supports TIB access, default hash codes, and locking; `AllocatorHeader`, which supports garbage collection information; and `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 RVMClass records the preferred alignment of the object as a whole. We lay out double and long fields first (and object references if these are 8 bytes long) so that we can avoid making holes in the field layout for alignment. We don't do this for smaller fields as all objects need to be a multiple of 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 section provides some explanation of how Java™ threads are scheduled and synchronized by Jikes™ RVM.

All Java threads (application threads, garbage collector threads, etc.) derive from `RVMThread`. These threads are multiplexed onto one or more virtual processors (see `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 a M:N thread model, scheduling execution of an arbitrarily large number (M) of Java threads over a finite number (N) of `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 th 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 in 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 `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 `ProcessorLocks`.

Transferring execution from one thread (A) to another (B) is a complex operation negotiated by the yield and morph methods of `RVMThread` and the dispatch method of `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 `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 `Lock`). Such migration will be inhibited if the thread is the last (non-idle) executable thread on its current virtual processor.

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

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

Synchronization

Jikes RVM uses a light-weight locking scheme to implement Java monitors (see `Lock` and `ThinLock`). The exact details of the locking scheme are dependent on which variant of 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 `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 `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 `RVMThread` invokes such an operation the whole `Processor` will be blocked and unable to schedule any other `RVMThreads`.

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 `ThreadIOQueue`. The `Processor` periodically checks pending IO operations and after they complete the calling thread is moved from the `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 Jul 02, 2008 by dgrove.

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

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

The appropriate interface names can be obtained by appending "Monitor" to the event names (e.g. the interface to implement for the MethodOverride event is Callbacks.MethodOverrideMonitor). Likewise, the method to register the callback is "add", followed by the name of the interface (e.g. the register method for the above interface is Callbacks.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 Controller class in the adaptive system and the 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 Callbacks interface. The specifics are specified below:

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

    ```
    import com.ibm.jikesrvm.Callbacks;
    ```

    b. before the call to runOnce add the following:

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

    c. after the call to runOnce add the following:

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

2. Recompile the modified file:

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

    or create a stub version of Callbacks.java and place it the appropriate directory structure with your modified file, i.e., com/ibm/jikesrvm/Callbacks.java
3. Run Jikes RVM as you normally would using the SPECjvm98 benchmarks.

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

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

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

This page last changed on Jul 02, 2008 by dgrove.

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 RVMThread 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/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 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 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/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/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 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 Jul 03, 2008 by dgrove.

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 attemtps at fully compiled language runtimes).

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

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

- Compiler Intrinsics: Most methods are within class librarys but some functions are built in (that is, intrinsic) to the compiler. These are referred to as intrinsic functions or intrinsics.
- Compiler Pragmas: Some intrinsics are do not provide any behaviour but instead provide information to the compiler that modifies optimizations, calling conventions and activation frame layout. We rever 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: Unfetted access to memory.
- Uninterruptible Code: Declaring code to be uninterruptible.
- Alternative Calling Conventions: Declaring different calling conventions and activation frame layout.

Compiler Intrinsics

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

Magic

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

A Jikes™ RVM implementor must be *extremely careful* when writing code that uses `Magic` to circumvent the Java type system. The use of `Magic.objectAsAddress` to perform various forms of pointer arithmetic is especially hazardous, since it can result in pointers being "lost" during garbage collection. All such uses of magic must either occur in uninterruptible methods or be guarded by calls to `VM.disableGC` and `VM.enableGC`. The optimizing compiler performs aggressive inlining and code motion, so not explicitly marking such dangerous regions in one of these two manners will lead to disaster.

Since magic is inexpressible in the Java programming language , it is unsurprising that the bodies of `Magic` methods are undefined. Instead, for each of these methods, the Java instructions to generate the code is stored in `GenerateMagic` and `GenerateMachineSpecificMagic` (to generate HIR) and `BaselineCompilerImpl` (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

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

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.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.

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 annotation 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 an 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.

MMTk

The garbage collectors for 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 semi spaces, a nursery space and a mature space, a set of generations, a non-relocatable buddy hierarchy etc depending upon the allocation and reclamation strategy employed by the memory manager.

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.

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

This page last changed on Mar 07, 2007 by pdonald.

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 frameworkby 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 `gcspyStartserver()` 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.gcspy` must be set to `1` such as in the `BaseBaseSemiSpaceGCspy`configuration. You can also have the Jikes RVM build process create a script to start the GCSpy client tool if GCSpy was built with support for client component. To achieve this the configuration parameter `config.include.gcspy-client` must be set to `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.gcspy-
client=1
```

It is also possible to build the Jikes RVM with GCSpy support but link it against a fake stub implementation rather than the real GCSpy implementation. This is achieved by setting the configuration parameter `config.include.gcspy-stub` to 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:gcspyPort=3000 -X:gc:gcspyWait=true &
```

Then you need to start the GCspy visualiser client.

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

After this you can specify the port and host to connect to (i.e. localhost:3000) and click the "Connect" button in the bottom right-hand corner of the visualiser.

Command line arguments

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

- `-X:gc:gcspyPort=<port>`
  The number of the port on which to connect to the visualiser. The default is port `0`, which signifies no connection.
- `-X:gc:gcspyWait=<true|false>`
  Whether Jikes RVM should wait for a visualiser to connect.
- `-X:gc:gcspyTilesize=<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 methods `traceObject` and `willNotMove` to ensure that tracing deals properly with GCspy objects: the GCspyTraceLocal file will be similar for any instrumented collector.

The instrumented collector, `SSGCspyCollector`, extends `SSCollector`. It needs to override `collectionPhase`.

Similarly, `SSGCspyMutator` extends `SSMutator` and must also override its parent's methods `collectionPhase`, to allow the allocators to collect data; and its `alloc` and `postAlloc` methods to allocate GCspy objects in GCspy's heap space.

The Plan

`SSGCspy.startGCspyServer` is called immediately before the "main" method is loaded and run. It initialises the GCspy server with the port on which to communicate, adds event names, instantiates a driver for each space, and then starts the server, forcing the VM to wait for a GCspy to connect if necessary. This method has the following responsibilities.

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 Jun 29, 2008 by rgarner.

This section describes the practical aspects of getting started using and modifying 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
     • The MMTk Test Harness
7. Debug the RVM
8. Evaluate the RVM

Building the RVM

This page last changed on Jul 23, 2008 by ianrogers.

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

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

Overview

Compiling the source code

The majority of Jikes RVM is written in Java and will be compiled into class files just as with other Java applications. There is also a small portion of Jikes RVM that is written in C that must be compiled with a C compiler such as gcc. Jikes RVM uses Ant version 1.6.5 or later as the build tool that orchestrates the build process and executes the steps required in building Jikes RVM.

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

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

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

Class libraries

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

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

Setting the ant property **classlib.provider** to **Harmony** (see how to define ant properties) will change the build process to download and build the Apache Harmony class library.

Target Requirements

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 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.

Jikes RVM currently supports the IA-32 ISA. As IA-32e is backward compatible with IA-32, Jikes RVM can be built and run upon IA-32e processors. The IA-64 architecture supports IA-32 code through a compatibility mode or through emulation and Jikes RVM should run in this configuration.

Operating Systems

Jikes RVM is capable of running on any operating system that is supported by the GNU Classpath library, low level library support is implemented and memory layout is defined. The low level library support includes interaction with the threading and signal libraries, memory management facilities and dynamic library loading services. The memory layout must also be known, as Jikes RVM will attempt to locate the boot image code and data at specific memory locations. These memory locations must not conflict with where the native compiler places it's code and data. Operating systems that are known to work include 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 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
ia32-solaris	Solaris	ia32	32 bits	OK
ppc32-aix	AIX	PowerPC	32 bits	OK
ppc32-linux	Linux	PowerPC	32 bits	OK
ppc32-osx	OSX	PowerPC	32 bits	OK
ppc64-aix	AIX	PowerPC	64 bits	OK
ppc64-linux	Linux	PowerPC	64 bits	Not Working
x86_64-linux	Linux	ia32	32 bits*	OK

- x86_64 is currently only supported using the legacy 32bit addressing mode and instructions.

Tool Requirements

Java Virtual Machine

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 recomended 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](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

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, Jikes RVM uses the bison tool which should be present on most modern *nix environments.

Perl

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

Awk

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

Extra tools recommended for Solaris

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

Instructions

Defining Ant properties

There are a number of ant properties that are used to control the build process of Jikes RVM. These properties may either be specified on the command line by "`-Dproperty=variable`" or they may be specified in a file named "`.ant.properties`" in the base directory of the jikesrvm source tree. The "`.ant.properties`" file is a standard Java proeprty 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 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 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 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 Jikes RVM.	""
components.dir	The directory where Ant looks for external components when building the RVM.	`${jikesrvm.dir}/components`
dist.dir	The directory where Ant stores the final Jikes RVM runtime.	`${jikesrvm.dir}/dist`
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)
components.cache.dir	The directory where Ant caches downloaded components. If you explicitly download a component, place it in this directory.	(Undefined, forcing download)

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 overide 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 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 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 just download and install the GNU Classpath class library you can run the command "`ant -f build/components/classpath.xml`". After this command has completed running it should have downloaded and

built the [GNU Classpath](#) class library for the current host. See the [Using GCSpy](#) page for directions on building configurations with GCSpy support.

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

Building the RVM

The next step in building Jikes RVM is to run the ant command "`ant`" or "`ant -Dconfig.name=...`". This should build a complete RVM runtime in the directory "${dist.dir}/${config.name}_${target.name}". 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 build and image directory for the current configuration.
very-clean	As with clean but also remove Java files generated by scripts.
real-clean	As with very-clean but remove all compiled images.
profiled-image	Compile a baseline version of the RVM for profiling then use the profile information to recompile the given `config.name` image.

Running the RVM

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

Building Patched Versions

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

The following steps will create a runtime without the patch in `dist/prototype_ia32-linux` and a runtime with the patch applied in `dist/prototype_ia32-linux_ReadBarriers`.

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

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

Cross-Platform Building

This page last changed on Jun 21, 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.

Dependencies

To compile the boot image on the host system you will also need to have built any dependencies on the target machine and then copied them to the host machine. You will also need to add an appropriate line into your *${components.dir}components.properties* file such as the following (if the target system was pppc32-linux).

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

⚠ **Note**

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

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

Primordial Class List

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

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.*) to support the above. This includes threading, memory management, runtime support for compiled code, etc.

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

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

This page last changed on Apr 07, 2008 by steveblackburn.

Overview

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

```
bin/buildit -h
```

...or read the script itself.

Examples

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

Building

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

```
bin/buildit habanero production
```

Or *equivalently*:

```
bin/buildit localhost production
```

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

```
bin/buildit jalapeno production
```

Cross Platform Building

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

```
bin/buildit chipotle production
```

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

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

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

Full Build Specification

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

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

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

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

which would build all three configurations on jalapeno.

Profiled Builds

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

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

Testing

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

Running a test-run

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

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

Running a test

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

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

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

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

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

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

Configuring the RVM

This page last changed on Apr 26, 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 *opt* or *base*.	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 *opt* or *base*.	base
config.include.aos	Include the adaptive system if set to true. Parameter will be ignored if config.runtime.compiler is not *opt*.	false
config.assertions	Parameter specifies the level of assertions in the code base. Must be one of *extreme*, *normal* or *none*.	normal
config.include.all-classes	Include all the Jikes RVM classes in the bootimage if set to true.	false
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 true to build RVM with GCSpy support. See Using GCSpy for more details.	false
config.include.gcspy-client	Set to true to bundle the GCSpy client with the Jikes RVM build. Parameter will be ignored if config.include.gcspy is not *true*.	false
config.include.gcspy-stub	Set to true to use the GCSpy stub rather than the real GCSpy component. Parameter will be ignored if config.include.gcspy is not *true*.	false
config.stress-gc-interval	The build will stress test the gc subsytem if set to a positive value. The value indicates the number of allocations between collections	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 a 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 24, 2008 by steveblackburn.

There are different tools for debugging Jikes RVM:

GDB

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

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

The dynamically created Java code doesn't provide GDB with the necessary symbol information for debugging. As some of the Java code is created in the boot image, it is possible to find the location of some Java methods and to break upon them. To 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.

Other Tools

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

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

For example, cachegrind can be invoked by

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

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

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

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 Jikes RVM based on the GNU Classpath implementation.

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.

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
```

Typically you might run a benchmark several times and choose the set of replay data that produced the best performance.

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
```

Measuring GC performance

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

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

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

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

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

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

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 product JVMs, we recommend you test your installation with the DaCapo benchmarks. We expect Jikes RVM performance to be very close to Sun's HotSpot 1.5 server running the DaCapo benchmarks (see our Nighlty DaCapo performance comparisionspage for the daily data).   Of course, running DaCapo well does not guarantee that Jikes RVM runs all codes well.

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

1. Jikes RVM start-up may be slow compared to the some product JVMs.
2. Remember that the non-adaptive configurations (-X:aos:enable_recompilation=false -X:aos:initial_compiler=opt) opt-compile *every* method the first time it executes. With aggressive

optimization levels, opt-compiling will severely slow down the first execution of each method. For many benchmarks, it is possible to test the quality of generated code by either running for several iterations and ignoring the first, or by building a warm-up period into the code. The SPEC benchmarks already use these strategies. The adaptive configuration does not have this problem; however, we cannot stipulate that the adaptive system will compete with the product on short-running codes of a few seconds.

3. 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 load balancing in the system is naive and unfair. This can hurt some styles of codes, including bulk-synchronous parallel programs.

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

Get The Source

The source code for Jikes RVM is stored in a Subversion repository. A developer can either work with the version control system or download one of the releases. If you are interested in doing development of Jikes RVM you may be interested in using a distributed or local version control tool.

Download a Release

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

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

or for the tar-bzip2 archive type:

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

Use Subversion

The source code for 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://jikesrvm.svn.sourceforge.net/svnroot/jikesrvm/rvmroot/trunk jikesrvm
```

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

```
> svn co https://jikesrvm.svn.sourceforge.net/svnroot/jikesrvm/rvmroot/tags/2.4.6 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/.

Using Distributed And Local Version Control Tools

This page last changed on May 21, 2008 by ianrogers.

Tools for Local and Distributed Version Control

1. Mercurial
2. GIT
3. SVK
4. DARCS
5. SVN Mirror
6. Rsync

Using Mercurial

Mercurial is a distributed version control tool (similar to Git) that is a bit cleaner to use and friendlier to Windows. Both systems allow keeping track of an SVN repository. However, while Git provides support for pushing back changes to the SVN repository, Mercurial does not. With mercurial, the way to publish changes is the generate patches and apply it to the SVN repository. As you will see soon, this is not too hard to do.

hgsvn is a third-party tool that support Mercurial-Subversion interoperability. It is available here: http://pypi.python.org/pypi/hgsvn

Set up a clone of the JikesRVM trunk by

```
$ hgimportsvn -r REV http://jikesrvm.svn.sourceforge.net/svnroot/jikesrvm/rvmroot/trunk
$ cd trunk
$ hgpullsvn
$ cd ..
```

This creates a directory named "trunk", which is a mercurial repository mirroring JikesRVM SVN trunk repository (starting at revision REV). It has a unique named branch "trunk". We will create a new directory "mywork" (and a separate branch "mywork") to make our changes to JikesRVM. With such a setup, the diff between the "mywork" branch and the "trunk" branch easily gives the patch to be committed to the JikesRVM Subversion trunk.

```
$ hg clone trunk mywork
$ cd mywork
$ hg branch mywork

  1. Do your work.
  2. Create new files using "hg add"
  3. Commit changes using "hg commit"
  4. Continue working ...
```

The differences between your work and the trunk can be seen with:

```
$ hg diff -r trunk > mywork.patch
```

It is possible that the JikesRVM trunk has in the meanwhile been updated. Get these changes by:

```
$ cd ../trunk
$ hgpullsvn
$ cd ../mywork
$ hg pull # pulls changes from the trunk repository to this directory.
$ hg merge
```

```
$ hg commit
$ hg diff -r trunk > mywork.latest.patch
```

An unofficial mercurial mirror of the JikesRVM trunk (synced twice a day) is maintained at http://www.cs.utexas.edu/users/speedway/jikesrvm-hg/trunk You can clone this mirror and eliminate some of the work shown above.

```
$ hg clone http://www.cs.utexas.edu/users/speedway/jikesrvm-hg/trunk mywork
$ cd mywork
$ hg branch mywork
```

1. Make changes and commit them
   $ hg commit
2. Synchronize with the Subversion HEAD
   $ hg pull; hg merge; hg commit
3. The changes you made against trunk can be got with
   $ hg diff -r trunk > mywork.patch

Using GIT to manage a local version of the Jikes RVM repository

GIT is a distributed version control tool that allows you to set up your own clone repositories of the Jikes RVM SVN repository that people can collaboratively work in. Changes can be pushed back to the Jikes RVM SVN repository or pulled from the Jikes RVM SVN repository.

Set up SVN clone

Set up a repository to hold a clone of the main Jikes RVM SVN repository (NB. we only set to import the trunk as bringing in everything else would be too slow):

```
> mkdir jikesrvm-svn.git
> cd jikesrvm-svn.git
> git svn init -T https://jikesrvm.svn.sourceforge.net/svnroot/jikesrvm/rvmroot/trunk
```

Bring in the Jikes RVM SVN (NB. you probably only want to pull in from a particular revision, if you don't specify -r and a version then the whole Jikes RVM history will be pulled across which will be very slow):

```
> git svn fetch -r 14100
```

Create a cloned local public repository to work off

Having a local clone of the SVN cloned repository will allow you to make branches that you can then push into the SVN clone which you can them commit back into the SVN repository:

```
> cd ..
> git clone --bare jikesrvm-svn.git jikesrvm.git
```

Checkout a local world to work in

```
> git clone jikesrvm.git myjikesrvm.git
```

Push a change back into your local public repository

First do a local commit. Git requires you to add what you want to commit and then commit, the -a option to commit can do these two actions together.

```
> cd myjikesrvm.git
> vi ...
> git commit -a
```

You then need to push this back to local public repository:

```
> git push
```

Push a change from the local public repository to the SVN repository clone

```
> cd ../jikesrvm.git
> git push <path to jikesrvm-svn.git> +master
```

The master is what is being pushed, the + forces the push.

Commit changes from the Jikes RVM SVN clone back to the Jikes RVM SVN repository

```
> cd ../jikesrvm-svn.git
> git svn dcommit
```

Pull changes into the Jikes RVM SVN clone from Jikes RVM SVN

```
> cd ../jikesrvm-svn.git
> git svn dcommit
```

Push changes from the Jikes RVM SVN clone to the local public Jikes RVM repository

```
> cd ../jikesrvm-svn.git
> git push ../jikesrvm.git +master
```

Bring local repository up-to-date with local public Jikes RVM repository

```
> cd ../myjikesrvm.git
> git pull
```

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://jikesrvm.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 //jikesrvm/local //jikesrvm/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 //jikesrvm/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 //jikesrvm/trunk //jikesrvm/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 //jikesrvm/trunk //jikesrvm/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 //jikesrvm/patchsets/news_add //jikesrvm/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/jikesrvm
> svk smerge --baseless -Il //jikesrvm/ //mirror/mymirror/jikesrvm
```

You can update the mirror at anytime via:

```
> svk smerge -I -l //jikesrvm/ //mirror/mymirror/jikesrvm
```

Use darcs

While a central SVN repository is excellent for collaboration on a common code base, sometimes you wish to try things before commiting 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 someting 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, inclusing 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 svm.

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:

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

```
> svm 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

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.

One goal of the JikesRVM project over recent years has been the ability to develop JikesRVM in a development environment such as Eclipse. This has been possible for the MMTk component since 2005, and as of early 2007 (release 2.9.0) it is possible to work with the majority of the JikesRVM codebase in Eclipse and similar environments. This is not yet as straightforward as we would like, and can be expected to improve with time.

- Editing JikesRVM in an IDE
- Compiler DNA
- Adding a New GC

This page last changed on Jun 30, 2008 by steveblackburn.

Overview

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

Prerequisites

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

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

Then test your GC:

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

You should have seen some output like this:

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

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

Cloning the MarkSweep GC

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

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

     c. Edit BaseBaseMyGC.properties, changing the text:
        "*config.mmtk.plan=org.mmtk.plan.marksweep.MS*" to
        "*config.mmtk.plan=org.mmtk.plan.**mygc**.MS*"
  3. Now test your new GC:

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

You should have got similar output to your test of MarkSweep above.

That's it. You're done. 🙂

Making it Prettier

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

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

Beyond BaseBaseMyGC

You probably want to build with configurations other than just BaseBase. If so, clone configurations from MarkSweep, just as you did above (for example, clone *FastAdaptiveMarkSweep* as *FastAdaptive**MyGC***).

What Next?

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

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

Coding Conventions

This page last changed on Jul 03, 2008 by dgrove.

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 Jul 03, 2008 by dgrove.

Regrettably, some code in the current system does not follow any consistent coding style. This is an unfortunate residuum of the system's evolution. To alleviate this problem, we present this style guide for new Java™ code; it's just a small tweak of Sun®'s style guide. We also use checkstyle to support a gradually expanding subset of these conventions. The current set of enforced checkstyle rules are defined by $RVM_ROOT/build/checkstyle/rvm-checks.xml and are verified as part of the pre-commit test run.

File Headers

Every file needs to have the license header.

A Java example of the notices follows.

```
/*
 *  This file is part of the Jikes RVM project (http://jikesrvm.org).
 *
 *  This file is licensed to You under the Common Public License (CPL);
 *  You may not use this file except in compliance with the License. You
 *  may obtain a copy of the License at
 *
 *      http://www.opensource.org/licenses/cpl1.0.php
 *
 *  See the COPYRIGHT.txt file distributed with this work for information
 *  regarding copyright ownership.
 */
package org.jikesrvm;

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

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

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 are to be no tabs in the source files or trailing white space on any line.
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 `Configuration` class and the `Properties` interface also are in that format. Since the `VM` class inherits fields from both `Properties` and `Configuration`, that's how we get `VM.VerifyAssertions`, etc.

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.

Compiler DNA

This page last changed on Jul 03, 2008 by dgrove.

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

1. run the compiler-dna test harness ("ant -f test.xml -Dtest-run.name=compiler-dna"). This will automatically compile and run Jikes RVM on SPEC JVM '98. You will want to configure the ant property external.lib.dir to be a directory containing your SPEC JVM '98 directory. Your SPEC JVM '98 directory must be named "SPECjvm98".
2. load the xml file "results/tests/compiler-dna/Report.xml" into either an XML viewer (such as a web browser) or into a text editor
3. find the section named *Measure_Compilation_Base*, then look within this section for statistics and find the static *Base.bcb/ms*. For example, '<statistic key="Base.bcb/ms" value="1069.66"/>'. In the compilationRates array this will be the value of element 0, it corresponds to how many bytecodes the baseline compiler can compile per millisecond.
4. find the section named *Measure_Compilation_Opt_0* and the statistic *Opt.bcb/ms*. This is element 1 in the compilationRates array.
5. find the section named *Measure_Compilation_Opt_1* and the statistic *Opt.bcb/ms*. This is element 2 in the compilationRates array.
6. find the section named *Measure_Compilation_Opt_2* and the statistic *Opt.bcb/ms*. This is element 3 in the compilationRates array.
7. find the section named *Measure_Performance_Base* and the statistic named *aggregate.best.score* and record its value. For example, for '<statistic key="aggregate.best.score" value="28.90"/>' you would record 28.90.
8. find the section named *Measure_Performance_Opt_0* and the statistic named *aggregate.best.score*. Divide this value by the value you recorded in step 7, this is the value for element 1 in the speedupRates array. For example, for '<statistic key="aggregate.best.score" value="137.50"/>' the speedupRates array element 1 should have a value of 4.76.
9. find the section named *Measure_Performance_Opt_1* and the statistic named *aggregate.best.score*. As with stage 8 divide this value by the value recorded in step 7, this is the value for element 2 in the speedupRates array.
10. find the section named *Measure_Performance_Opt_2* and the statistic named *aggregate.best.score*. As with stage 8 divide this value by the value recorded in step 7, this is the value for element 3 in the speedupRates array.

You should then save CompilerDNA and recompile a production RVM which will use these values.

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

Editing JikesRVM in an IDE

This page last changed on Jul 08, 2008 by steveblackburn.

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

Editing JikesRVM in Eclipse

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

1. Create a JikesRVM source tree either via SVN checkout or untar-ing a distribution.

   ```
   $ svn co https://jikesrvm.svn.sf.net/svnroot/jikesrvm/rvmroot/trunk jikesrvm
   ```

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

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

     Note that if you will not or cannot build on your local machine, substitute localhost for the name of a host you can build on (buildit will perform the build remotely and then copy the requisite files back).
   - If you are working on an **older version** (**2.9.1 - 2.9.3**), you can follow this procedure:

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

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

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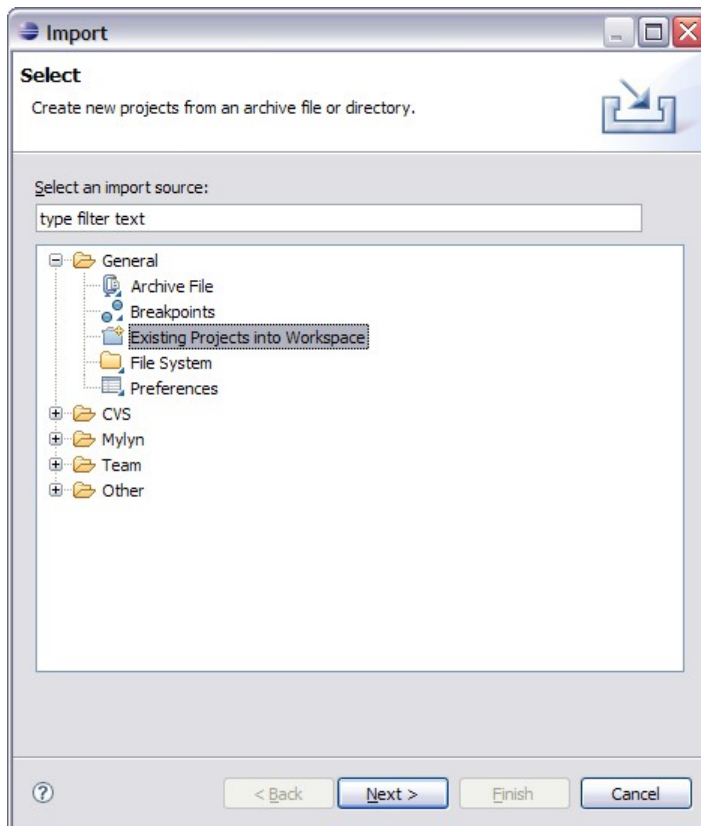
c. Browse to find the top-level directory.
d. Select the project (in this case JikesRVM ia32-linux development)
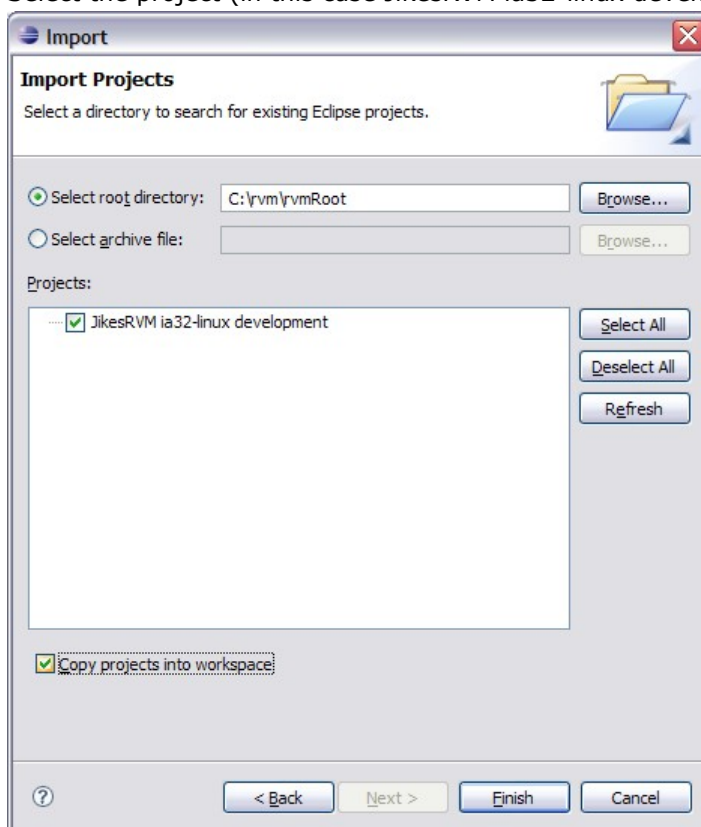
e. Hit Finish

Editing JikesRVM in NetBeans

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

Profiling Applications with Jikes RVM

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

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

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

In addition to the sampling-based mechanisms, the baseline compiler can inject code to gather branch probabilites 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 `AOSDatabase.debuggingCounterData.getCounterInstructionForEvent(String eventName);`. This method returns an `Instruction` that can be inserted into the code. The instruction will increment a counter associated with the String name "eventName", and the counter will be printed at the end of execution.

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

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

Managed Data:

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

Instrumentation Phases:

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

Lower Instrumentation Phase:

This phase converts the high level "count event" instruction into the actual counter code by using the counter manager. It currently occurs at the end of LIR, so instrumentation can not be inserted using this mechanism after LIR. This phase does not need to be modified if you add a new phase, except that the

`shouldPerform()` method needs to have your instrumentation listed, so this phase is run when your instrumentation is turned on.

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 appopriate configuration
5. ./dist/prototype-opt_ia32-linux/rvm -version   # Change dir to use selected host and configuration

Running the RVM

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 java.lang.instrument

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

These options are all prefixed by -X:gc:.

Boolean options.

Option	Description
protectOnRelease	Should memory be protected on release?
echoOptions	Echo when options are set?
printPhaseStats	When printing statistics, should statistics for each gc-mutator phase be printed?
xmlStats	Print end-of-run statistics in XML format
eagerCompleteSweep	Should we eagerly finish sweeping at the start of a collection
fragmentationStats	Should we print fragmentation statistics for the free list allocator?
verboseFragmentationStats	Should we print verbose fragmentation statistics for the free list allocator?
verboseTiming	Should we display detailed breakdown of where GC time is spent?
noFinalizer	Should finalization be disabled?
noReferenceTypes	Should reference type processing be disabled?
fullHeapSystemGC	Should a major GC be performed when a system GC is triggered?
ignoreSystemGC	Should we ignore calls to java.lang.System.gc?
variableSizeHeap	Should we shrink/grow the heap to adjust to application working set?
eagerMmapSpaces	If true, all spaces are eagerly demand zero mmapped at boot time
sanityCheck	Perform sanity checks before and after each collection?

Value options.

Option	Type	Description
markSweepMarkBits	int	Number of bits to use for the header cycle of mark sweep spaces
verbose	int	GC verbosity level
stressFactor	bytes	Force a collection after this much allocation
metaDataLimit	bytes	Trigger a GC if the meta data volume grows to this limit
boundedNursery	bytes	Bound the maximum size of the nursery to this value
fixedNursery	bytes	Fix the minimum and maximum size of the nursery to this value
debugAddress	address	Specify an address at runtime for use in debugging

Base Compiler Non-Standard Command-Line Options

Boolean options

Option	Description

Option	Description
edge_counters	Insert edge counters on all bytecode-level conditional branches
invocation_counters	Select methods for optimized recompilation by using invocation counters

Opt Compiler Non-Standard Command-Line Options

Boolean options.

Option	Description
local_constant_prop	Perform local constant propagation
local_copy_prop	Perform local copy propagation
local_cse	Perform local common subexpression elimination
global_bounds_check	Perform global Array Bound Check elimination on Demand
monitor_removal	Try to remove unnecessary monitor operations
invokee_thread_local	Compile the method assuming the invokee is thread-local
no_callee_exceptions	Assert that any callee of this compiled method will not throw exceptions?
simple_escape_ipa	Eagerly compute method summaries for simple escape analysis
field_analysis	Eagerly compute method summaries for flow-insensitive field analysis
scalar_replace_aggregates	Perform scalar replacement of aggregates
reorder_code	Reorder basic blocks for improved locality and branch prediction
reorder_code_ph	Reorder basic blocks using Pettis and Hansen Algo2
inline_new	Inline allocation of scalars and arrays
inline_write_barrier	Inline write barriers for generational collectors
inline	Inline statically resolvable calls
guarded_inline	Guarded inlining of non-final virtual calls
guarded_inline_interface	Speculatively inline non-final interface calls
static_splitting	CFG splitting to create hot traces based on static heuristics
redundant_branch_elimination	Eliminate redundant conditional branches
preex_inline	Pre-existence based inlining
ssa	Should SSA form be constructed on the HIR?
load_elimination	Should we perform redundant load elimination during SSA pass?
coalesce_after_ssa	Should we coalesce move instructions after leaving SSA?
expression_folding	Should we try to fold expressions with constants in SSA form?
live_range_splitting	Split live ranges using LIR SSA pass?
gcp	Perform global code placement
gcse	Perform global code placement
verbose_gcp	Perform noisy global code placement
licm_ignore_pei	Assume PEIs do not throw or state is not observable
turn_whiles_into_until	Turn whiles into untils
loop_versioning	Loop versioning
handler_liveness	Store liveness for handlers to improve dependence graph at PEIs
schedule_prepass	Perform prepass instruction scheduling
no_checkcast	Should all checkcast operations be (unsafely) eliminated?
no_checkstore	Should all checkstore operations be (unsafely) eliminated?
no_bounds_check	Should all bounds check operations be (unsafely) eliminated?
no_null_check	Should all null check operations be (unsafely) eliminated?
no_synchro	Should all synchronization operations be (unsafely) eliminated?

no_threads	Should all yield points be (unsafely) eliminated?
no_cache_flush	Should cache flush instructions (PowerPC SYNC/ISYNC) be omitted? NOTE: Cannot be correctly changed via the command line!
reads_kill	Should we constrain optimizations by enforcing reads-kill?
monitor_nop	Should we treat all monitorenter/monitorexit bytecodes as nops?
static_stats	Should we dump out compile-time statistics for basic blocks?
code_patch_nop	Should all patch point be (unsafely) eliminated (at initial HIR)?
instrumentation_sampling	Perform code transformation to sample instrumentation code.
no_duplication	When performing inst. sampling, should it be done without duplicating code?
processor_specific_counter	Should there be one CBS counter per processor for SMP performance?
remove_yp_from_checking	Should yieldpoints be removed from the checking code (requires finite sample interval).

Value options.

Option	Description
ic_max_target_size	Static inlining heuristic: Upper bound on callee size
ic_max_inline_depth	Static inlining heuristic: Upper bound on depth of inlining
ic_max_always_inline_target_size	Static inlining heuristic: Always inline callees of this size or smaller
ic_massive_method_size	Static inlining heuristic: If root method is already this big, then only inline trivial methods
ai_max_target_size	Adaptive inlining heuristic: Upper bound on callee size
ai_min_callsite_fraction	Adaptive inlining heuristc: Minimum fraction of callsite distribution for guarded inlining of a callee
edge_count_input_file	Input file of edge counter profile data
inling_guard	Selection of guard mechanism for inlined virtual calls that cannot be statically bound
fp_mode	Selection of strictness level for floating point computations
exclude	Exclude methods from being opt compiled
unroll_log	Unroll loops. Duplicates the loop body 2^n times.
cond_move_cutoff	How many extra instructions will we insert in order to remove a conditional branch?
load_elimination_rounds	How many rounds of redundant load elimination will we attempt?
alloc_advice_sites	Read allocation advice attributes for all classes from this file
frequency_strategy	How to compute block and edge frequencies?
spill_cost_estimate	Selection of spilling heuristic
infrequent_threshold	Cumulative threshold which defines the set of infrequent basic blocks
cbs_hotness	Threshold at which a conditional branch is considered to be skewed
ir_print_level	Only print IR compiled above this level

Adaptive System Non-Standard Command-Line Options

Boolean options

Option	Description
enable_recompilation	Should the adaptive system recompile hot methods?
enable_advice_generation	Do we need to generate advice file?

enable_precompile	Should the adaptive system precompile all methods given in the advice file before the user thread is started?
enable_replay_compile	Should the adaptive system use the pseudo-adaptive system that solely relies on the advice file?
gather_profile_data	Should profile data be gathered and reported at the end of the run?
adaptive_inling	Should we use adaptive feedback-directed inlining?
early_exit	Should AOS exit when the controller clock reaches early_exit_value?
osr_promotion	Should AOS promote baseline-compiled methods to opt?
background_recompilation	Should recompilation be done on a background thread or on next invocation?
insert_yieldpoint_counters	Insert instrumentation in opt recompiled code to count yieldpoints executed?
insert_method_counters_opt	Insert intrusive method counters in opt recompiled code?
insert_instruction_counters	Insert counters on all instructions in opt recompiled code?
insert_debugging_counters	Enable easy insertion of (debugging) counters in opt recompiled code.
report_interrupt_stats	Report stats related to timer interrupts and AOS listeners on exit.
disable_recompile_all_methods	Disable the ability for an app to request all methods to be recompiled.

Value options

Option	Description
method_sample_size	How many timer ticks of method samples to take before reporting method hotness to controller.
initial_compiler	Selection of initial compiler.
recompilation_strategy	Selection of mechanism for identifying methods for optimizing recompilation.
method_listener_trigger	What triggers us to take a method sample?
call_graph_listener_trigger	What triggers us to take a method sample?
logfile_name	Name of log file.
compilation_advice_file_output	Name of advice file.
dynamic_call_file_output	Name of dynamic call graph file.
compiler_dna_file	Name of compiler DNA file (no name ==> use default DNA). Discussed in a comment at the head of VM_CompilerDNA.java.
compiler_advice_file_input	File containing information about the methods to Opt compile.
dynamic_call_file_input	File containing information about the hot call sites.
logging_level	Control amount of event logging (larger ==> more).
final_report_level	Control amount of info reported on exit (larger ==> more).
decay_frequency	After how many clock ticks should we decay.
dcg_decay_rate	What factor should we decay call graph edges hotness by.
dcg_sample_size	After how many timer interrupts do we update the weights in the dynamic call graph?
ai_seed_multipiler	Initial edge weight of call graph is set to ai_seed_multipler * (1/ai_control_point).
offline_inline_plan_name	Name of offline inline plan to be read and used for inlining.
early_exit_time	Value of controller clock at which AOS should exit if early_exit is true.
invocation_count_threshold	Invocation count at which a baseline compiled method should be recompiled.
invocation_count_opt_level	Opt level for recompilation in invocation count based system.

counter_based_sample_interval	What is the sample interval for counter-based sampling.
ai_hot_callsite_threshold	What percentage of the total weight of the dcg demarcates warm/hot edges.
max_opt_level	The maximum optimization level to enable.

Virtual Machine Non-Standard Command-Line Options

Boolean Options

Option	Description
measureCompilation	Time all compilations and report on exit.
measureCompilationPhases	Time all compilation sub-phases and report on exit.
stackTraceFull	Stack traces to consist of VM and application frames.
stackTraceAtExit	Dump a stack trace (via VM.syswrite) upon exit.
verboseTraceClassLoading	More detailed tracing then -verbose:class.
errorsFatal	Exit when non-fatal errors are detected; used for regression testing.

Value options

Option	Description
maxSystemTroubleRecursionDepth	If we get deeper than this in one of the System Trouble functions, try to die.
interruptQuantum	Timer interrupt scheduling quantum in ms.
schedulingMultiplier	Scheduling quantum = interruptQuantum * schedulingMultiplier.
traceThreadScheduling	Trace actions taken by thread scheduling.
verboseStackTracePeriod	Trace every nth time a stack trace is created.
edgeCounterFile	Input file of edge counter profile data.
CBSCallSamplesPerTick	How many CBS call samples (Prologue/Epilogue) should we take per time tick.
CBSCallSampleStride	Stride between each CBS call sample (Prologue/ Epilogue) within a sampling window.
CBSMethodSamplesPerTick	How many CBS method samples (any yieldpoint) should we take per time tick.
CBSMethodSampleStride	Stride between each CBS method sample (any yieldpoint) within a sampling window.

Running Jikes RVM with valgrind

Jikes RVM can run under valgrind, as of SVN revision 6791 (29-Aug-2007). Applying a patch of this revision to release 3.2.1 should also produce a working system. Versions of valgrind CVS prior to release 3.0 are also known to have worked.

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

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

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

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

Testing the RVM

This page last changed on Apr 06, 2008 by steveblackburn.

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 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:irc: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.

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

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

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

	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.	
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 *test-configuration*.	""
configuration	The name of the RVM build configuration to test.	<tag>
target	The name of the RVM build target. This can be used to trigger compilation of a profiled image	"main"
mode	The test mode. May modify the way test groups execute. See individual groups for details.	""
extra.args	Extra arguments that are passed to the RVM.	""
extra.rvm.args	Extra arguments that are passed to the RVM. These may be varied for different runs using the same image.	""

ⓘ **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}
```

Test Specific Parameters

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

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

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

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

Excluding tests

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

i.e. 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 "test.config.basic.TestSuspend.exclude=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 `${results.dir}/tests/${test-run.name}/Report.html` to get an overview on test run or at `${results.dir}/tests/${test-run.name}/Report.xml` for an xml document describing test results.

External Test Resources

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

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

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

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

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???...
```

Test Run Descriptions

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

Red test run

This test run **MUST** be run prior to committing code. They are relatively short and are designed to capture as many potential bugs in the shortest possible time.It is expected that the red test run will take 15-20 minutes on modern intel architecture.

Green test run

There is a set of workloads we consider important (i.e. dacapo and SPEC*). There is a set of build configurations we consider important (ie prototype, development, production). We as a group wish to guarantee that all important workloads will will run correctly on all important build configurations. (i.e. We should **NEVER** regress). The green test run is designed to identify as early as possible any failures in this matrix of build configuration x workload. It is run continuously 24 hours a day (or at least every time a change is made). It is expected that the green test run will take 2-6 hours to complete depending on the environment.

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

Blue test run

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

Rainbow test runs

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

Summary

We must **NEVER** regress in green test run. The red test run attempts to ensure no green regressions this while keeping running time reasonable. The blue test run gives us an overall picture on the health of the code base. While the rainbow test runs are used at different times for different purposes.

This page last changed on Jun 30, 2008 by steveblackburn.

Overview

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

Running the test harness

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

Standalone

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

There is a collection of sample scripts in the MMTk/harness/test-scripts directory.

In Eclipse

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

Define a new run configuration with main class org.mmtk.harness.Main.

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

Test harness options

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

Option	Meaning
plan	The MMTk plan class. Defaults to org.mmtk.plan.marksweep.MS
collectors	The number of concurrent collector threads (default: 1)
initHeap	Initial heap size
maxHeap	Maximum heap size (default: 64 pages)
trace	Debugging messages from the MMTk Harness. Trace options include • CALL - trace procedure calls • ALLOC - trace object allocation • OBJECT - trace object mutation events
gcEvery	Force frequent GCs. Options are • ALLOC - GC after every object allocation • SAFEPOINT - GC at every GC safepoint

Scripting language

Basics

The language has two types: integer and object. Objects are allocated
with the 'alloc' statement, and have a specified number of pointers and
nonpointers (integers). Variables are declared 'c' style, and are optionally
initialized at declaration.

Object fields are referenced using syntax like "tmp.int5" or "tmp.objecti*3",
ie like a struct of arrays of the appropriate types.

Syntax

```
script ::= method...

method ::= ident "(" { type ident { "," type ident}...  ")" "{" statement... "}"

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

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

type ::= "int" | "object"

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

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

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

unop ::= "!" | "-"
```