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 `org.jikesrvm.aos` package.

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

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.