Adaptive Optimization System

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

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

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

Selective Optimization

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

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

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