The JikesRVM Adaptive Optimization System
Adaptive Compilation (aka Adaptive Optimization)

- Compiling at the method level is
  - Slow – much slower than cost of interpreting one instruction
  - Optimizing compiler (as efficient as it is) is very high overhead

- If we compile everything
  - Big startup delay
  - Big delay the first time we execute a method

- Goal: combine interpretation and compilation to get the best of both
  - Interpreting first: fast startup, no long pauses
  - Identify the frequently executing methods (hot methods)
  - Compile them (with some optimization) in the background
    - Execute them the next time around
Multi-compiler (Mixed Mode) System

- **Compile-only vs Compile+Interpret strategy**
  - Baseline – (could be replaced with interpretation) ...
    - **Simulates** execution using the bytecode and operand stack
    - Translates bytecodes to native code directly
    - No optimization, **no register allocation**
    - Performance much like an interpreter
    - **Fast compilation/interpretation, SLOW code**
  - Optimizing
    - Translates bytecodes to HIR->LIR->MIR
    - Optimization is performed on each level
    - Linear scan register allocation
    - **Slow compilation/fast code**
JikesRVM Compiler Differences

- Compile Time/Speed comparison
- 500MHz RS6000, 4GB Mem, 1-processor

**Compile time**: Bytecode bytes per millisecond
- Baseline: 378, L0: 9.3, L1: 5.7, L2: 1.8

**Code speed** normalized to baseline
- L0: 4.3, L1: 6.1, L2: 6.6

**EX**: L2 is 209 times slower to compile & produces code that is 6.6 times faster
JikesRVM Threading

- Two alternatives
  - Native threads: Map each Java thread to an OS pthread; OS-managed
    - Less work for the runtime (simpler) for scheduling
    - More work for the runtime to facilitate GC (since thread switching can now happen on any instruction)
      - Compiler generates GC maps (list of roots) at every instruction
  - Green threads: Java threads are multiplexed on virtual processors; JVM/runtime managed in coordination with OS
    - A virtual processor is an OS pthread
    - Require software support for switching (yielding the processor so that other threads can take a turn) – *yield points*
    - Compiler generates this support
      - Generates GC maps (list of roots) at every yield point
JikesRVM Threading

- Java threads are multiplexed on virtual processors
  - A virtual processor is an OS pthread
- Yield points
  - Compiler generated
  - Points in a method where a thread **checks** to see if it should give up the processor (\& give another thread a turn)
    - Check a bit in a register, if set then call scheduler
    - Set is caused by **timer** interrupt
  - **Method prologues**
  - **Back edges of loops**

```java
x = 20
L1: if x>=10 goto L3
    ...
goto L1
L3: y = x + 5

x = 20
goto L1
L0: yeild
L1: if x>=10 goto L3
    ...
goto L0
L3: y = x + 5
```
Adaptive Optimization System (AOS) Architecture

Primary components:

- Runtime measurements subsystem
- Controller
- Recompilation System
- Database
Adaptive Optimization System Architecture
Runtime Measurements Subsystem

- Gathers information about executing methods
- Summarizes the information
- Passes the summary to the event system
- Records the summary in a database
Runtime Measurements Subsystem

- Gathers information about executing methods
- Summarizes the information
- Passes the summary to the event system
- Records the summary in a database

Information (Profiling!)

- From the VM
  - When it performs services for the program (thread switch, memory allocation, compilation, garbage collection, etc.)
- From instrumentation
  - Code added to the executing methods
  - Methods in application and VM
  - Invocation counters, edge, path, value profiling
- Hardware performance counters
  - Cache misses (instruction/data)
Runtime Measurements Subsystem

- Information is stored in raw format

**Organizers**

- Threads that periodically process the information, analyze it, and format it appropriately for use by the controller
- Separates data generation from analysis
  - Multiple organizers can process the same data (in different ways)
  - Profiling code can then operate under rigid resource constraints
    - Example: VM memory allocator profiler
    - Can’t allocate memory
    - Should complete quickly so as not to interrupt execution
  - Overlap analysis with application execution
Controller

- Manages the adaptive optimization system
- Coordinates activities of runtime measurement subsystem and the recompilation system
- Initiates all profiling activity by determining what profiling
  - Should occur
  - Under what conditions
  - For how long
- Gets its information from the runtime measurement subsystem and the AOS database
- Passes compilation decisions to the recompilation subsystem (continue or change)
  - Inserts them into a compilation queue as a “compilation plan”
Sampling to Identify Hot Methods

- To estimate the time spent in a method
- Sample on yield points only (ie \textit{when} a thread yields)
  - Yield points: @ method entries (prologue) & loop back edges

- Before \textbf{switching} threads, a counter associated with the method that is executing (current) is incremented
  - When a loop backedge is traversed a counter is incr’ed.
  - When a method prologue is entered
    - A counter for the invoked method is incremented
    - A counter for the calling method is incremented

- This information (and HW counter information) is stored as raw data
Sampling

- Three threads access the raw data
  - Method listener object (created by the hot method organzr)
    - On each thread switch, records the currently active method in the raw data buffer – **runs on the application thread**
    - Wakes hot method organizer after sample size has been reached
  - Hot method organizer
    - Scans the method counter raw data to identify methods in which the most time is spent – **in the background**
    - “hot” if the percentage of samples attributed to that method exceeds a controller-directed threshold
      - And the method is not already compiled to maximum degree
    - Enqueues an event in the event Q for each hot method (and %age)
  - Decay organizer – **decrements method counters** (in bg)
    - Gives more weight to recent samples (for hotness identification)
Recompilation

- Given a hot method, the controller decides if it is profitable to recompile a method
  - Cost model
    - Expected time the method will execute if not recompiled
    - Cost for recompiling the method at a certain optimization level
    - Expected time the method will execute if recompiled
  - Goal: minimize the expected cost for the method in the future
  - Assumptions made: future execution time, code performance, recompilation time ...
Recompilation

• Assumptions are made for all expected values
  - Program will execute for twice the duration that it has
    - Uses samples to estimate percentage of program time spent in the method in question
  - Offline measurements indicate the effectiveness of each optimization level
    - How much faster the method will run
• Cost of recompilation
  - Linear model of compilation speed for each optimization level as a function of method size.
  - Calibrated offline
Recompilation Subsystem

- Multiple threads that invoke compilers
  - extract and execute compilation plans in compilation queue
    - plans are inserted into queue by controller
- Controller provides them with a *compilation plan*
  - Instrumentation Plan
  - Profile Data – for feedback-directed optimizations
  - Optimization Plan
More on Program Profiling

Questions?
Profiling

• “the formal summary or analysis of data representing distinctive features or characteristics” (*American Heritage Dictionary*)
  - Program profiling: analyzing the execution characteristics of code to extract and summarize its behavior

• Offline vs online
  - Offline – Collection time doesn’t matter
  - Online – Slow-down is an important factor

• Exhaustive vs sample-based
  - Sampling is estimation. The difference between a sampled profile and an exhaustive profile is the accuracy/error measure
  - Sampling is commonly used online where time matters
    - Can be used offline if inaccuracy can be tolerated
Why Profile?

- To characterize program behavior
  - Understand how programs behave
  - Guide tool, runtime, system (hw/sw) design
  - Program test generation

- To capture specific or unusual behavior
  - Security attacks, intrusion detection, bugs, test coverage
  - Logging

- Track performance regressions
Why Profile?

- To improve performance
  - Time different parts of the program to find out where time is being spent
    - 80/20 rule – identify the 20 and focus your optimization energy
    - By hand optimization
    - Automatic (compiler or runtime) feedback-directed optimization
      - Target hot code
      - Inlining and unrolling
      - Code scheduling and register allocation

- Increasingly important for speculative optimization
  - Hardware trends → simplicity & multiple contexts
  - Less speculation in hardware, more in software
What to Profile

- Individual instructions
  - Memory accesses (allocations/deletions, loads/stores)
    - Lends insight into caching, paging, GC, races, bugs & more
  - If individual instruction detail isn’t needed: capture basic blocks
    - Estimate bb’s by recording branches and their direction
    - E.g: lends insight into branch miss overhead

- Paths

- Function invocations and call sites

- Memory allocation, GC time

- Interfaces (ABI, APIs to other components, foreign function)

- Resource use
  - CPU, Network, disk, other I/O
  - Runtime services (compiler/interp, GC, runtime, OS)

- User interactivity
Instrumentation vs Event Monitoring

- Instrumentation: Insert code into the code of a program
  - The additional code executes interleaved with program code
  - To collect information about the program code activity
- Can perturb the behavior that it is trying to measure
Instrumentation vs Event Monitoring

- **Instrumentation**: Insert code into the code of a program
  - The additional code executes interleaved with program code
  - To collect information about the program code activity

- **Event monitoring**
  - Profiling external to the executing program
  - Output timestamps, upon OS or runtime activity, around program
  - Record of operations (timings, counts) in runtime that execute concurrently with the executing program, yet independent of it
    - Garbage collection activity
    - Accesses to the OS
    - Accesses to libraries (e.g. GUI, cloud SDK, web services)
  - **Hardware performance counters/monitors (HPMs)**

- Can perturb the behavior that it is trying to measure
Hardware Performance Monitors/Counters

- Libraries provide access to hardware collected HPMs

- Other types of sampling
  - Random
  - Periodic
  - Phase
Program Behavior changes over time

- Different behavior during different parts of execution

- Many programs execute as a series of phases possibly with recurring patterns

- Capture via basic block profiles for fixed number of instructions=vector
  - Compare counts across vectors for similarity
Sampling Interactive Sessions

- A period of user interaction: Each application has a specific pattern

Interactivity Session - Tetrix
Interactive Sessions

- A period of user interaction: Each application has a specific pattern

Interactivity Session - Solitaire
Advantages of the profiling framework (listed in paper)

- Instrumentation can be performed longer without degrading performance
- Tunable
- Existing instrumentation techniques can be incorporated without modification
- Multiple types of instrumentation can be used simultaneously
- Does not require hardware support
- Framework is deterministic so easier to debug

Downsides/limitations

- JikesRVM-specific
Sample-based Instrumentation

- Turn on and off instrumentation dynamically
  - Challenge: when to turn instrumentation on and off
  - Why is this important to do?

- How: **Switching between (un-)instrumented versions**
  - Via **code patching**: Ephemeral Instrumentation, DynInst, IBM Java Developer Kit
    - Have two versions of the methods (or code blocks) you want to instrument
    - In the uninstrumented version, put a patch point at entry
      - Dummy instruction large enough to hold a jump
    - Overwrite (patch) the entry point to instrument
    - “Undo” patch to turn off instrumentation
  - Via **recompilation and on-stack replacement**

- Via **code copying**
(Arnold-Ryder PLDI'01)
Advanced Instrumentation-based Program Profiling

▪ If we have time...
Profiling Tools

• Of binaries (independent of language)
  - Pin, Dynamo
  - Valgrind
  - gprof (call graph and function timings)

• Of programs (language specific)
  - Java – JVMPITI, JProfiler, many others, GCSpy
  - Ruby – ruby-prof
  - Python – cprofile

• HPMs
  - Library support: PAPI
  - OS Integration: PerfMon, OProfile, XenOProf
Exhaustive Path Profiling (Instrumentation)

Thanks to Mike Bond (Ohio State) for his presentations of PEP and Continuous Path/Edge Profiling for these slides [CGO/MICRO 2005] on path profiling and its optimization.

- Processors need long instruction sequences
- Programs have branches
Why path profiling?

- Compiler identifies hot paths across multiple basic blocks
Why path profiling?

- Compiler identifies hot paths across multiple basic blocks
  - Forms and optimizes “traces” -- like superinstructions!
Why path profiling?

- Compiler identifies hot paths across multiple basic blocks
  - Speculative in that control may exit the trace early
    - Requires special case handling/overhead
Ball-Larus path profiling

- 4 paths $\rightarrow [0, 3]$
Ball-Larus path profiling

- 4 paths $\rightarrow [0, 3]$
- Each path sums to a unique integer
Ball-Larus path profiling

- 4 paths $\rightarrow [0, 3]$
- Each path sums to unique integer

Path 0
Ball-Larus path profiling

- 4 paths $\rightarrow [0, 3]$
- Each path sums to unique integer

Path 0

Path 1
Ball-Larus path profiling

- 4 paths $\rightarrow [0, 3]$
- Each path sums to unique integer

Path 0
Path 1
Path 2
Ball-Larus path profiling

- 4 paths $\rightarrow [0, 3]$
- Each path sums to unique integer

Path 0
Path 1
Path 2
Path 3
Ball-Larus path profiling

- \( r \): path register
  - Computes path number

- \texttt{count}:
  - Stores path frequencies

\[ r = r + 1 \]
\[ r = r + 2 \]
\[ \text{count}[r]++ \]
Updates path profile
Ball-Larus path profiling

- $r$: path register
  - Computes path number

- `count`:
  - Stores path frequencies
  - Array by default
  - Too many paths?
    - Hash table
    - High overhead

\[
\begin{align*}
  & r = 0 \\
  & r = r + 1 \\
  & r = r + 2 \\
  & \text{count}[r]++ \\
  & \text{Updates path profile}
\end{align*}
\]
Optimizing Path Profiling

- Where have all the cycles gone?

```
r = r + 1
```

- cheap <10%

- expensive >90%

- \( r = 0 \)

- \( r = r + 1 \)

- \( r = r + 1 \)

- count\[r\]++
Optimizing Path Profiling

All-the-time instrumentation

Sampling (piggybacks on existing mechanism)

SAMPLE r

r=0

r=r+1

r=r+2
Optimizing Path Profiling

All-the-time instrumentation

Overhead: 30% → 2%

[Bond et al. 2005]

Sampling (piggybacks on existing mechanism)

SAMPLE r
Profile-guided profiling

- Existing edge profile informs path profiling
  - Profile some initially
    - Quite fast to profile edges
    - Can be sample based
    - Just need to determine which branch edges are taken more frequently
Profile-guided profiling

- Existing edge profile informs path profiling
- Assign zero to hotter edges
  - No instrumentation