Profiling Java Applications Using Code Hotswapping and Dynamic Call Graph Revelation

Mikhail Dmitriev

Sun Microsystems Lab

Presented by

Ophelia Chesley

Instrumentation-based Profiling

- Advantage:
  - Reveal dynamic behavior of modern software
  - Gathers low-level (CPU, memory usages) data as well as high-level data (GUI events, EJB security)
- Disadvantage:
  - Incurs high overhead – excess time to execute
  - Skews cache misses
  - Prevents certain optimizations
  - Static instrumentation stays with the target application during the entire run
- Proposed Solution:
  - Profile only limited subsets of methods
  - On-line instrumentation

Implemented Solution: JFluid

- JFluid Server with target VM
  - Receives and execute commands from users
  - Inform users of events pertaining to the target application
  - Transmit profiled data to users
  - Minimal communication and profiler code
- JFluid GUI Tool
  - Use the ProfileServer class to start the target VM, or
  - attach JFluid to the running VM using the UNIX signal SIGQUIT
  - Processes rough profiling data and builds compact profiling results

Hotswapping

- Only methods, not classes are modified/instrumented
- Locate all pointers to old method versions
- Create new method versions in parallel with other Java threads
- Suspend all Java threads
- Deoptimize methods that were previously inlined/compiled (provided by Hotspot VM)
- Switch pointers to the respective new method objects
- Resume all application threads

Dynamic Instrumentation of a Call Subgraph

- User select an arbitrary method root for instrumentation
- JFluid tool will recursively:
  - Scan an executing instrumented method (m) to find next called method (VC.v) to instrument
  - Check loaded subclasses of VC to find any methods v that may overrides VC.v and instrument them by hotswapping
  - Check each newly loaded class whether any of its methods override the instrumented methods of its superclasses

Results

- Currently only support collection of CPU profiling data
- Compare overhead between fully profiled versus partially profiled target applications (SPECjvm98 and PetStore)
- For small benchmarks, partial profiling still results in considerable overhead (breaks optimization)
- Partial profiling in large benchmarks incurs 2-12% overhead
- For benchmarks with many polymorphic calls, dynamic instrumentation still results in many instrumented methods that are not called – wasted time for hotswapping.