The Jalapeno Dynamic Optimizing Compiler for Java

IBM, Java Grande 99

http://www.research.ibm.com/jalapeno

Implementing Jalapeno in Java. - OOPSLA'99
Jalapeno - a Compiler-supported Java Virtual Machine for Servers. - WCSSS 99
Overview

- Review Terminology
- Overview of Jalapeno VM
  - All components
- Details of Jalapeno Optimizing Compiler
- Performance Results
Java Terminology

- **Java Virtual Machine**
  - Handles all aspects of executing Java bytecodes
    - Execution
    - Memory management (garbage collection)
    - Thread Scheduling

- **Tools used by a JVM**
  - Interpreter
  - JIT compiler
  - Dynamic compiler
Example Java VM’s

Matt’s JVM
- Interpret bytecodes

Nasko’s JVM
- Dynamic compiler
- Profiling support
- Advanced garbage collector
Overview of Jalapeno VM

- Written entirely in Java
- Primary goal:
  - High performance and scalability of Java applications on SMP server machines
- Compile only approach
  - no interpretation
  - Mixing optimized and un-optimized is easy
- Supports all features of Java
  - dynamic class loading, reflection, etc
Components of Jalapeno VM

- Three compilers
  - Baseline
  - Quick
  - OPT

- Adaptive Optimization System
  - Profiler (Online measurements)
  - Controller (the brain)

- Object allocator

- Variety of type accurate garbage collectors
  - copying / non copying
  - generational / non generational

- Thread scheduler, dynamic loader, etc
Implementation in Java

- Written entirely in java
  - All subsystems
  - Self bootstrapping
    - Does not run on another JVM
    - Creates “boot image”
      - Contains compiled version of all code needed to start VM running
      - Must be able to load and compile methods
    - Self optimization occurs during execution
Advantages

- Runs along side the application code
- Layers
  - User code
  - Java library code
  - VM
  - Operating System
  - Hardware
- No barrier between VM, library, and user code
  - No Language switching
  - Better optimization
    - For example: Can inline allocation code into user code
Jalapeno Optimizing Compiler

- Job of optimizing compiler
  - Input:
    - Set of methods
    - Time budget
  - Output:
    - Executable code for methods

- Can operate as static compiler
  - Include application code in boot image
Optimizing Compiler Internal Structure

- HIR
  - High level IR
  - VM and machine independent
  - Different from bytecode
    - Register based
    - Explicit exception checks

- LIR
  - Low level IR
  - VM specific, machine independent

- Mir
  - Machine IR
  - Machine specific
Jalapeno Front End

- BC2IR (bytecodes -> HIR)
  - Essentially interprets bytecodes
  - Maintains symbolic state
    - Corresponds to abstract values of stack operands & locals
  - Constructs CFG
  - Performs on the fly
    - Dead code, copy propagation
      - Eliminates most temporaries
  - Exploits
    - At a control flow join, values of stack operands may differ, but the types of these operands must match
Jalapeno Back End

- Lowering the IR (HIR -> LIR)
  - Becomes VM specific
    - invokestatic becomes specific VM call

- Dependence Graph Construction
  - LCPC 99 paper
    - Compute all memory and register dependences
    - Linear
    - Considers all of Java’s constructs
      - Synchronization
      - Exceptions
Jalapeno Back End (continued)

- LIR -> MIR
  - BURS tree pattern matching system

- Register Allocation
  - Supports different register allocators
    - Based on time budget
  - Current:
    - Greedy linear scan, not graph coloring
    - Several Times faster
    - Works almost as well
Optimizations

- Flow-sensitive optimizations
  - Topic of future work

- Flow Insensitive
  - Within a basic block
    - CSE, redundant check removal
  - One trick:
    - Every variable must have a value before it’s used
    - If a variable has only one definition, it reaches all uses.
    - Can build def-use chains and perform
      - copy propagation, dead code elimination
    - Not as good as flow sensitive
Performance

- Compare
  - JDK w/o JIT (1.1.6)
  - JDK w/ JIT (IBM enhanced port)
  - Jalapeno Baseline
  - Jalapeno Optimized

- Compile times NOT included
  - Run tests multiple times, exclude first run
Performance

- Conclusions
  - Micro Benchmarks
    - Jalapeno OPT
      - Better on 3 benchmarks
      - Within 1.6 on the rest
  - Macro Benchmarks
    - 1.2 to 3.3 times slower than product JIT
- Very Encouraging
  - Happy to be within order of magnitude
  - Jalapeno still very immature
  - Product JIT performs many optimization not yet implemented in Jalapeno
Work in Progress

- Interprocedural Analysis
  - Optimization of register restore/save
  - Escape Analysis (oopsla 99)
  - Handling dynamic class loading
Experiences Implementing with Java

- Weakness in the Java language
  - Mismatch between Java’s compilation strategy and the make system
    - Only way to ensure all files are up to date
      - make clean; make
  - Limited package mechanism
    - No equivalent of a C++ friend
    - Can’t divide up JVM subcomponents into packages
    - Major namespace problems
  - Lack of pre-processor support
    - Cannot conditionally include fields and methods in class definition