An Analysis of Specialized Bytecodes:
Do Specialized Bytecodes Have Value?

Ben Stephenson
Wade Holst
University of Western Ontario
Outline

- Background
- Motivation
- Despecializations
- Specializations
- Performance Results
- Conclusion
Background

• Java Bytecodes are a representation used to express a Java program
  – Core Bytecodes: provide unique functionality
  – Specialized Bytecodes:
    • Functionality for some bytecodes is duplicated exactly by another bytecode
    • Some others can easily be mimicked with a short sequence of other bytecodes
Background

• The JVM Specification defines the functionality of 201 bytecodes
  – Examples: aload_0, invokevirtual, ifne, …
  – 55 bytecodes remain whose functionality is undefined by the standard

• Some VMs use the undefined bytecodes
  – Optimizations in software implementations
  – Additional functionality in hardware implementations
Motivation

- JVM Specification defines 201 bytecodes
  - Leftover bytecodes are already being used for implementation specific functionality
- It may be desirable to create new Java bytecodes to
  - Support new language features efficiently
  - Support for new native types
  - Introduce new bytecodes that allow common operations to be performed more efficiently
Despecializations

- Despecializations remove redundant bytecodes by expressing their functionality with other equivalent bytecodes
  - Consider the bytecode `iload_1`
    - Loads local variable at index 1 onto the stack as an integer
  - Functionality is duplicated by the `iload` bytecode
    - Loads an arbitrary local variable onto the stack as an integer
### Despecializations

<table>
<thead>
<tr>
<th></th>
<th>Before:</th>
<th></th>
<th>After:</th>
</tr>
</thead>
<tbody>
<tr>
<td>n-1</td>
<td>...</td>
<td>n-1</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>iload_1</td>
<td>n</td>
<td>iload</td>
</tr>
<tr>
<td>n+1</td>
<td>bytecode</td>
<td>n+1</td>
<td>1</td>
</tr>
<tr>
<td>n+2</td>
<td></td>
<td>n+2</td>
<td>bytecode</td>
</tr>
</tbody>
</table>
Despecializations

• Load Despecializations
  – \texttt{<type>load\_<n>} → \texttt{<type>load} (20 cases)

• Store Despecializations
  – \texttt{<type>store\_<n>} → \texttt{<type>store} (20 cases)

• Constant Despecialization
  – \texttt{<type>const\_<n>} → \texttt{ldc[2][\_w]/bipsuh} (14 cases)

• Additional Despecializations
  – Widen constants, branch transformations (13 cases)
Specializations

• Introduce a new bytecode to replace instances of a core bytecode with a specific argument
  – `<type>load` → `<type>load_<n>`
  – `<type>store` → `<type>store_<n>`
  – `bipush/sipush/ldc[2][_w]` → `<type>const_<n>`

• Some core bytecodes with a specific argument are more common than current specialized bytecodes
Profiling Results

Frequency of Execution of Core and Specialized Bytecodes

- Core Bytecodes
- Specialized Bytecodes

Bar chart showing the percentage of bytecodes executed, with frequencies ranging from 0 to 3.5.
Performance Results

• Testing was performed on
  – Kaffe VM Interpreter 1.0.7

• Test machine
  – Pentium III
  – RedHat Linux version 9, kernel 2.4.20-8
Benchmarks

- SPEC JVM98 Suite
- Java Grande Forum
- Linpack
- Scimark
- Ashes
Performance Test Conditions

- Despecialize all specialized bytecodes that occur less than 0.1 percent of the time (28 despecializations)
- Specialize all core bytecodes with specific arguments that occur at least 0.1 percent of the time (42 specialization)
- Switch infrequently used specialized bytecodes with frequently occurring core bytecodes (31 switches)
Performance Results

- Most benchmarks showed small differences
- Average difference across all benchmarks was less than 1.0 percent for all 3 conditions
- Repeated Measures Analysis of Variance concluded that differences are not statistically significant at $p$ less than 0.05
Future Work

- Consider impact on class file size
  - Despecialization increases class file size by about 6 percent for SPEC benchmarks
  - No data collected for specialization / switch
- Measure performance impact on other Java Virtual Machine Implementations
- Use despecialized spaces for something more useful
Conclusions

• Existing specialized bytecodes have little impact on application runtime
• Picking more frequently occurring specialized bytecodes does not improve application performance
• Careful consideration should be given
  – to removing specialized bytecodes from JVM
  – by anyone developing a new VM
Questions?