An Evaluation of Specialized Java Bytecodes

Ben Stephenson and Wade Holst
University of Western Ontario, London, Ontario, Canada
ben@csd.uwo.ca wade@csd.uwo.ca

Abstract

Specialized Java bytecodes provide functionality that is easily replicated using other Java bytecodes. This study uses profiling to explore how the set of specialized bytecodes currently implemented by the Java Virtual Machine is utilized by comparing it to the other specialized bytecodes which could have been implemented.

1 Introduction

The Java Virtual Machine Specification defines the functionality of 201 distinct bytecodes. These bytecodes are used to represent the methods within a Java class file. At runtime, the bytecodes are executed, either using an interpreter or a just-in-time compiler. This study examines these bytecodes by categorizing them as either *general purpose* or *specialized*. The utilization of current Java bytecodes is examined through profiling, a new alternative set of specialized bytecodes is proposed, and the suitability of each set of specialized bytecodes is evaluated by examining its impact on class file size and benchmark performance.

2 Specialized Bytecodes

Java bytecodes can be classified as general purpose or specialized.

General Purpose: Bytecodes that provide unique functionality that cannot easily be replicated by one, or a short sequence of other bytecodes.

Specialized: Bytecodes that provide functionality that is easily replicated by one bytecode, or a short sequence of bytecodes.

Examples:

- `aload_0` is equivalent to `aload 0x00`
- `istore_3` is equivalent to `istore 0x03`
- `iconst_2` is equivalent to `bipush 0x02`
- `ifle` is equivalent to `iconst_0 if_icmple`

This study considers specialized load and store bytecodes in detail.

3 Benchmarks & Profiling

The following benchmarks are considered in this study:

- SPEC JVM98 Benchmark Suite
- Java Grande Forum Benchmarks
- Ashes Suite Collection

Each benchmark was executed on a modified implementation of the Kaffe virtual machine which recorded every bytecode executed, along with its operands. Performance testing was conducted on the distribution and modified versions of the Kaffe virtual machine using its interpreter and JIT3 execution engines.

4 Local Variable Loads

Local variables can be loaded onto the operand stack using both specialized and general purpose bytecodes. Figure 1 shows the most frequently executed load bytecodes. Specialized bytecodes are shown in blue while general purpose bytecodes are shown in yellow.

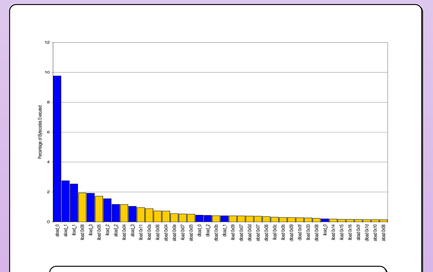

Figure 1: Load Bytecodes Executed with Greatest Frequency

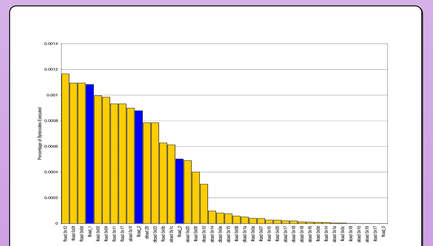

Figure 2: Load Bytecodes Executed with Least Frequency

The load bytecodes executed with least frequency are shown in Figure 2. We note that four specialized load bytecodes are present in this list, including `fload_0`, which is never executed by any of the benchmarks tested.

Why are some specialized load bytecodes executed infrequently? Why are some general purpose load bytecodes executed with great frequency?

Type Symmetry: An equal number of bytecodes are devoted to lead each of the 5 basic types supported by the JVM (object, double, float, int, and long). However, integer operations are performed with greatest frequency.

Receiver Object Parameter: The receiver object is always passed as the first parameter to instance methods. Consequently, loads of non-object type never occur from slot 0 in an instance method.

5 Local Variable Stores

Like loads, stores from the operand stack to a local variable can be performed with both general purpose and specialized bytecodes. Figure 3 shows the distribution of store bytecodes executed with greatest frequency.

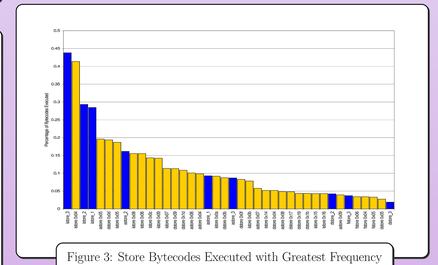

Figure 3: Store Bytecodes Executed with Greatest Frequency

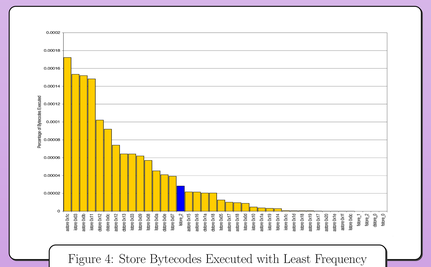

Figure 4: Store Bytecodes Executed with Least Frequency

Specialized store bytecodes are also found among the least executed store bytecodes. In particular, we observe that 5 specialized store bytecodes are present, including 4 which are never executed by any of the benchmarks profiled as part of this study.

Why are some specialized store bytecodes executed infrequently? Why are some general purpose store bytecodes executed with great frequency?

Pass By Value Method Parameters: All method parameters are passed using pass by value semantics. In addition, method parameters are stored in the first local variable slots. As a result, slots for which specialized store bytecodes exist are frequently used to hold method parameters. However, any change to a method parameter is not visible in the calling scope. Thus, stores to method parameters are uncommon, making the use of specialized store bytecodes uncommon.

Receiver Object Pointer Invariance: The Java Virtual Machine Specification specifically prohibits changing the pointer to the receiver object within an instance method. Consequently, none of the `<t>store_0` bytecodes can every execute within an instance method.

6 Constant Loads

Specialized bytecodes are also used to load constant values onto the operand stack. The most frequently used constant values are shown in Figure 5.

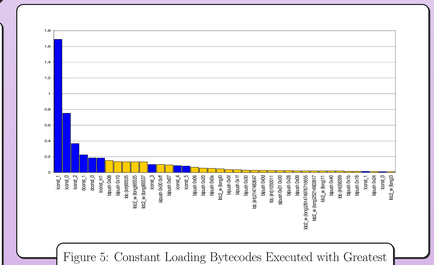

Figure 5: Constant Loading Bytecodes Executed with Greatest Frequency

The six most frequently used constants are loaded onto the operand stack using specialized bytecodes, which is a better use of specialized bytecodes than for either loads or stores. Furthermore, there are only 14 specialized constant loading bytecodes compared to 20 specialized loads and stores.

Why are specialized constant loading bytecodes better utilized than specialized load and store bytecodes?

Type Symmetry: Specialized load and store bytecodes are type symmetric – an equal number of bytecodes are allocated to each of the 5 primitive types supported by the Java Virtual Machine. The specialized constant loading bytecodes are not type symmetric. Half of the bytecodes are allocated to loading integer constants while the other half handle constants of type float, long and double.

Load / Store Symmetry: An equal number of specialized bytecodes are allocated to loads and stores. However, the number of loads performed is much larger than the number of stores. This suggests that more specialized bytecodes should be allocated for handling stores than loads.

7 Alternative Specialized Bytecodes

An alternative set of specialized load and store bytecodes, determined by profiling the benchmarks, is shown below. It is asymmetric with respect to both the data types specialized and number of load and store bytecodes.

Root Indices	
aload	0 1 2 3 4 5 7 0xE
dload	0 1 2 6 7 8 9 0xB 0xD 0xF
iload	0 1 2 3 4 5 6 7 8 9 0xA 0xB 0xC 0x11 0x14 0x33
astore	5
dstore	1
istore	1 2 3 4

8 Impact on Class File Size

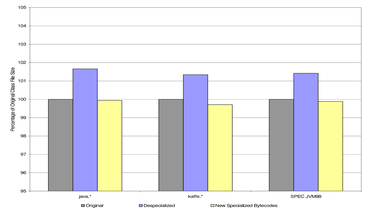

Figure 6: Change in Class File Size for the Library and Benchmark Classes

Performing despecialization increased average class file size by less than 1.7 percent in all cases. Using the new alternative set of specialized bytecodes decreased class file size. However, the reduction was less than 0.3 percent for each category considered.

9 Impact on Bytecodes Executed

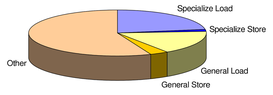

Figure 7: Distribution of Bytecodes Executed using the Specialized Bytecodes Defined in the Java Virtual Machine Specification

Figure 7 shows the distribution of the bytecodes executed by the JVM using the current symmetric set of specialized bytecodes. The distribution of bytecodes executed using the alternative set is shown in Figure 8. Using the alternative set of bytecodes has increased the specialized load and store bytecodes from 24 percent to 38 percent of all bytecodes executed.

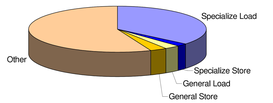

Figure 8: Distribution of Bytecodes Executed using the Alternative Specialized Bytecodes

10 Performance Impact

Performance measurements were gathered for six of the benchmarks profiled earlier in this study. Testing revealed that none of the benchmarks showed a large change in performance as a result of either despecialization or using our new, alternative set of specialized bytecodes.

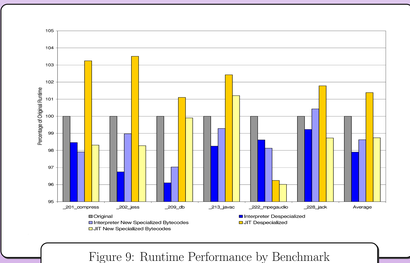

Figure 9: Runtime Performance by Benchmark

On average, the interpreter experienced a speedup for both despecialization and the alternative set of specialized bytecodes. However, this performance change was less than 2.1 percent on average, and less than 4.1 percent for any one benchmark.

When a JIT compiler was employed, performing despecialization resulted in a performance loss of 1.4 percent on average while using our alternative set of specialized bytecodes resulted in a minor performance gain of 1.3 percent on average.

11 Conclusion

This study examined the impact specialized bytecodes have on class file size and application performance. It considered both the removal of existing specialized load and store bytecodes, and the introduction of a new set of specialized load and store bytecodes identified using profiling.

Each of these changes had a small impact on class file size. Performing despecialization increased average class file size by 1.5 percent, while using the alternative set of specialized bytecodes decreased class file size by an average of 0.1 percent.

Application performance was also tested. While a mixture of improvements and losses were observed, all of the changes were of very small magnitude. This leads us to conclude that specialized bytecodes are contributing very little to the Java Virtual Machine. Consequently:

- When revisions are made to the Java Virtual Machine Specification, specialized load and store bytecodes should be considered for removal to make space for bytecodes that offer greater performance or class file gains, or to reduce the complexity of the virtual machine.
- Those involved in the creation of new intermediate languages should carefully consider whether specialized bytecodes will provide a benefit before including specialized bytecodes in their specification.