The Swift Java Compiler


Copyright 2015, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Citation numbers refer to entries in the EaC2e bibliography.
Background

**Swift was an attempt to build a serious optimizing compiler for Java**

- Translates Java bytecode into optimized assembly code for the **DEC Alpha**
- Alpha was a 64-bit **RISC** machine intended to replace the **VAX-11**
  - Design goal was high-frequency operation, enabled by manual chip design & layout

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>No branch delay slots</td>
<td>32 Integer &amp; 32 FP registers</td>
</tr>
<tr>
<td>No condition codes</td>
<td>IEEE FP &amp; VAX FP</td>
</tr>
<tr>
<td>No byte-oriented loads/stores</td>
<td>On-chip L1 &amp; memory controller</td>
</tr>
</tbody>
</table>

- Alpha (& Swift) were developed at **DEC WRL**, later Compaq **WRL**, later **HP**

- This compiler did not see much daylight. It is different than the other “classic” compilers that we will study. We study it because of the paper.
  - Good summary of state of the field as of 2000
  - Compiler targets generic Java code, rather than Fortran or PL.8
  - Nice evaluation methodology
Optimizing Java

Several characteristics of Java make optimization more difficult

• Heap allocation of all objects
• Synchronization in library routines (unnecessary in single threaded code)
• Virtual methods (slow runtime & complicate analysis)
• Required runtime checks
  ♦ Performing the checks slows the code
  ♦ Failing a check can raise an exception (more complications)
Optimizing Java

Several characteristics of Java make optimization more difficult

• Heap allocation of all objects
• Synchronization in library routines (unnecessary in single threaded code)
• Virtual methods (slow runtime & complicate analysis)
• Required runtime checks
  ♦ Performing the checks slows the code
  ♦ Failing a check can raise an exception (more complications)

Several features of Java make it an attractive target for optimization

• Strong typing eliminates many ambiguities found in other languages
  ♦ Local variables are unambiguous, as are fields of objects
• No unrestricted pointers, no pointer arithmetic
• Standard classes written in Java, so they can be optimized, too
This compiler is a full-blown optimizing compiler, rather than a JIT.

It translates Java methods from standard Java bytecodes into Alpha code, using a large suite of analyses and transformations.

This compiler is more complex than a typical JIT.

- Ambitious whole method and cross-method optimization
- Some mechanism to preserve information across compilations
Swift Compiler’s IR

Swift has a multi-level IR (similar to Fortran H and PL.8)

- Operations represented in Static Single Assignment Form
  - They discuss SSA as a graph; it can just as easily be viewed as a linear form
  - Each node in the graph represents an SSA name (or value)
    - Node has an operation & operand (edges to other nodes)
  - A set of nodes has an implicit partial order from the definition-use relationship

- Several kinds of ops
  - Simple arithmetic ops
  - Abstract ops such as phi, field accessors, allocation, invocation, various checks
  - Low-level, machine-dependent ops that map directly to Alpha ops (100 or so)

- Bytecode to IR pass builds SSA
  - Performs some local optimizations

- Lowering pass translates into low-level ops, when appropriate
  - Performs logical peephole optimization over edges in SSA graph

Rice University, COMP 512
The Swift IR includes a Control-Flow Graph (CFG)

- Nodes in the **CFG** represent basic blocks
  - Maximal length sequences of straight line code — a set of **SSA** value nodes
  - Each block ends with a transfer of control, including exception behavior
  - Block has a *control value* that determines whether or not it executes
    - Encodes simple notion of control dependence — blocks are partially ordered, too
- Edges in the **CFG** represent transfers of control
  - Edges for both normal transfers and exception-triggered transfers
- Swift breaks all critical edges to simplify later optimizations

---

AD is a critical edge
- A has multiple successors
- D has multiple predecessors
To break AD, insert an empty block at mid-edge
Swift Java IR’s Memory Model

In Swift Java IR, every SSA name is a local, unambiguous value
- Edges between use & def encode precise dependences for local scalars
- Global variables and heap-allocated objects may be ambiguous
  - Swift represents these values with explicit read and write operations
    - Fields of class objects or instance objects
    - Array elements
  - Compiler must maintain relative ordering of the reads and writes
    - Cannot move definition of a location past a read of that location (in either direction)
- To represent this constraint, Swift introduces a global store
  - Write operation takes global store as operand and produces a new one as result
  - Read operation takes global store as operand
  - Effectively serializes the store operations by threading them together in the SSA
  - Anti-dependences must be enforced by the scheduler
- The authors emphasize the IR memory model
The Problem with Stores & Loads

In general, a compiler must maintain the ordering of loads and stores implied in the source code, unless it can prove that the memory accessed by the reordered stores is disjoint.

- **SSA** does not have an edge that connects two memory operations that access the same location
  - If they use the same **SSA** value as the address, they are transitively connected
  - If they recompute the address, they are not connected

- Load-store & store-load order matter; load-load order does not
  - The compiler must maintain the serial order of stores
  - The compiler cannot move a load past a store, in either direction

- The Swift compiler introduces a global store to enforce true dependences
  - Writes consume a global store and produce a new one
  - Reads consume a global store
  - The **SSA** edges on the store enforce the correct order of memory operations
## Analysis and Optimization

They implemented a large set of analyses and transformations

<table>
<thead>
<tr>
<th>Interprocedural Analyses</th>
<th>Interprocedural Opt</th>
<th>Machine Dependent Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alias Analysis</td>
<td>Bound check removal</td>
<td>Lower IR</td>
</tr>
<tr>
<td>Class Hierarchy Analysis</td>
<td>Branch removal</td>
<td>Peephole optimizations</td>
</tr>
<tr>
<td>Escape Analysis</td>
<td>Constant propagation</td>
<td>Sign-extension elimination</td>
</tr>
<tr>
<td>Field Analysis</td>
<td>Dead code elimination</td>
<td>Trace scheduling</td>
</tr>
<tr>
<td>Type Propagation</td>
<td>Global CSE</td>
<td>Register allocation</td>
</tr>
<tr>
<td>Interprocedural Opt</td>
<td>Global code motion</td>
<td>Block layout</td>
</tr>
<tr>
<td>Method resolution</td>
<td>Loop peeling</td>
<td>Final code generation</td>
</tr>
<tr>
<td>Method inlining</td>
<td>Null check removal</td>
<td></td>
</tr>
<tr>
<td>Method splitting</td>
<td>Peephole optimization</td>
<td></td>
</tr>
<tr>
<td>Object inlining</td>
<td>Strength reduction</td>
<td></td>
</tr>
<tr>
<td>Stack allocation</td>
<td>Type test elimination</td>
<td></td>
</tr>
<tr>
<td>Synchronization removal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Experimental Evaluation

Platform

• Alpha 21264 Processor at 667 MHz
  • Separate 64 KB L1 I & D caches, 4 MB unified, off-chip, L2 cache
• Workstation running Compaq/DEC version of Unix (Tru64 Unix)
• High-performance JVM with a “quite good” JIT

Benchmarks

• SpecJVM98 plus others
• Compared execution times, in seconds, under a variety of scenarios

Compile time

• Swift compiles at 1800 to 2200 SLOC per second on the Alpha
  • That is without escape analysis (+ sync removal & stack allocation)
  • Those features slow down compilation by 20 to 40%
## Experimental Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Problem Domain</th>
<th>SLOCs</th>
<th>JVM Time (secs)</th>
<th>Swift Run Time (secs) w/o CHA</th>
<th>w/s-CHA</th>
<th>w/CHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>text compression</td>
<td>910</td>
<td>12.68</td>
<td>9.61</td>
<td>8.72</td>
<td>9.66</td>
</tr>
<tr>
<td>jess</td>
<td>expert system</td>
<td>9734</td>
<td>4.97</td>
<td>4.35</td>
<td>4.17</td>
<td>4.12</td>
</tr>
<tr>
<td>cst</td>
<td>data structures</td>
<td>1800</td>
<td>8.02</td>
<td>5.97</td>
<td>5.65</td>
<td>5.38</td>
</tr>
<tr>
<td>db</td>
<td>database retrieval</td>
<td>1026</td>
<td>17.73</td>
<td>15.62</td>
<td>12.73</td>
<td>12.44</td>
</tr>
<tr>
<td>si</td>
<td>interpreter</td>
<td>1707</td>
<td>8.09</td>
<td>6.48</td>
<td>5.93</td>
<td>6.33</td>
</tr>
<tr>
<td>javac</td>
<td>Java compiler</td>
<td>~18000</td>
<td>5.80</td>
<td>7.57</td>
<td>7.14</td>
<td>7.00</td>
</tr>
<tr>
<td>mpeg</td>
<td>audio decompressor</td>
<td>~3600</td>
<td>10.63</td>
<td>5.74</td>
<td>5.60</td>
<td>5.68</td>
</tr>
<tr>
<td>Richards</td>
<td>task queues</td>
<td>3637</td>
<td>8.09</td>
<td>8.52</td>
<td>5.30</td>
<td>4.69</td>
</tr>
<tr>
<td>mtrt</td>
<td>ray tracing</td>
<td>3952</td>
<td>4.69</td>
<td>5.11</td>
<td>2.09</td>
<td>1.59</td>
</tr>
<tr>
<td>jack</td>
<td>parser generators</td>
<td>~7500</td>
<td>5.92</td>
<td>5.27</td>
<td>4.90</td>
<td>4.96</td>
</tr>
<tr>
<td>tsgp</td>
<td>genetic programming</td>
<td>894</td>
<td>35.89</td>
<td>25.70</td>
<td>24.10</td>
<td>24.05</td>
</tr>
<tr>
<td>jlex</td>
<td>scanner generator</td>
<td>7569</td>
<td>4.96</td>
<td>4.10</td>
<td>3.84</td>
<td>2.95</td>
</tr>
</tbody>
</table>

**Speedup over JVM**

1.21  1.43  1.52

**JVM** is their optimized JVM running bytecode. Swift times are compiled code, loaded into their optimized JVM.

Rice University, COMP 512
Experimental Evaluation

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>inl</th>
<th>cha</th>
<th>fld</th>
<th>objinl</th>
<th>split</th>
<th>stk</th>
<th>sync</th>
<th>sr</th>
<th>cse</th>
<th>gcm</th>
<th>peel</th>
<th>ckelim</th>
<th>selim</th>
<th>br</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>1.16</td>
<td>1.20</td>
<td>1.16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.04</td>
</tr>
<tr>
<td>jess</td>
<td>1.07</td>
<td>1.09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cst</td>
<td>1.08</td>
<td>1.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.03</td>
</tr>
<tr>
<td>db</td>
<td>1.05</td>
<td>1.26</td>
<td>1.04</td>
<td>1.03</td>
<td>1.03</td>
<td>1.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>si</td>
<td>1.27</td>
<td>1.14</td>
<td></td>
<td>1.04</td>
<td>1.04</td>
<td>1.06</td>
<td>1.16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javac</td>
<td>1.09</td>
<td>1.09</td>
<td></td>
<td>1.05</td>
<td>1.04</td>
<td>1.06</td>
<td>1.16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.11</td>
</tr>
<tr>
<td>mpeg</td>
<td>1.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.05</td>
<td>1.35</td>
<td></td>
</tr>
<tr>
<td>richards</td>
<td>1.40</td>
<td>1.76</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.11</td>
</tr>
<tr>
<td>mtrt</td>
<td>1.57</td>
<td>2.68</td>
<td>1.27</td>
<td>1.16</td>
<td>1.13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ts gp</td>
<td>1.03</td>
<td>1.05</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.05</td>
</tr>
<tr>
<td>j lex</td>
<td>1.22</td>
<td>1.19</td>
<td>1.15</td>
<td>1.18</td>
<td>1.15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Entries represent the percent slowdown, from the Swift Runtime with CHA number in Table 1, when the optimization corresponding to that column is disabled.

Table 2 from the paper
Take-Away Points

• Optimizing Java requires some different analyses, but the compiler looks quite similar to Fortran H and PL.8

• IR design has a large influence on how well the compiler works
  ♦ You must represent it to optimize it!

• Decent selection of algorithms and techniques

• Interesting evaluation method
  ♦ Subtracting optimizations from the full set to see their impact
  ♦ Different results than you might see in an additive test
  ♦ Multiple transformations might catch the same effect (e.g., GCSE & code motion)