COMP 512
Rice University
Spring 2015

# *The Swift Java Compiler*

Daniel J. Scales, Keith H. Randall, Sanjay Ghemawat, and Jeff Dean, “The Swift Java Compiler: Design and Implementation”, COMPAQ WRL Research Report 2000/2, April 2000.

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

| | |
|---|---|
| No branch delay slots | 32 Integer & 32 FP registers |
| No condition codes | IEEE FP & VAX FP |
| No byte-oriented loads/stores | On-chip L1 & memory controller |

- ♦ 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 <u>all</u> 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 <u>all</u> 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*)

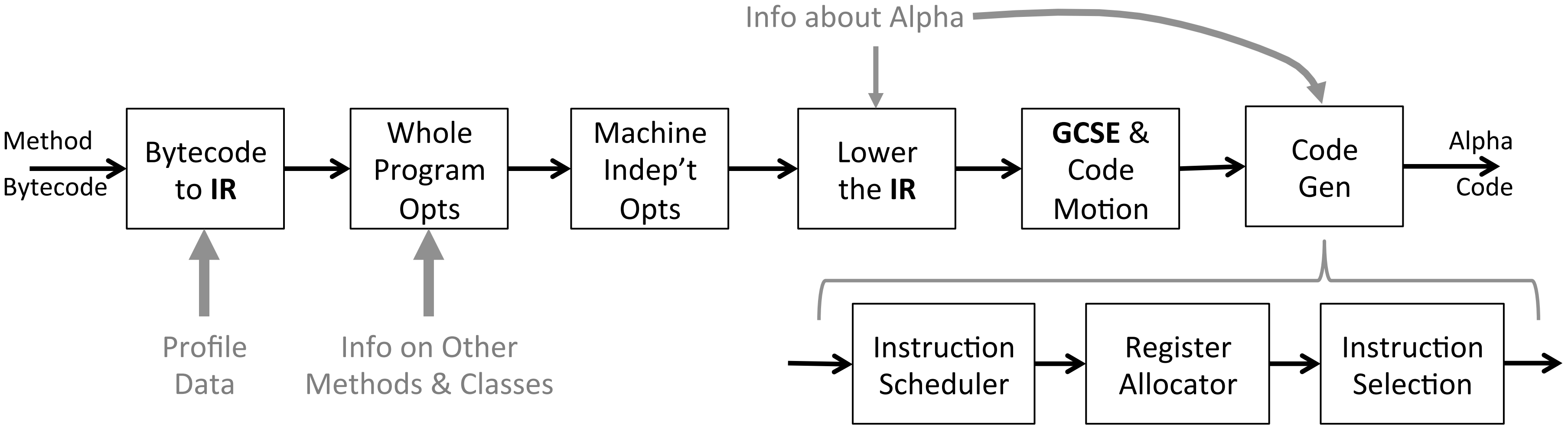
## Several features of Java make it an attractive target for optimization

- 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

# Compiler Structure

**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.



Classic Compiler Structure

**This compiler is more complex than a typical JIT.**

- Ambitious whole method and cross-method optimization
- Some mechanism to preserve information across compilations

Some kind of repository or "cache"

# Swift Compiler’s IR

## Swift has a multi-level IR

(*similar to Fortran H and PL.8*)

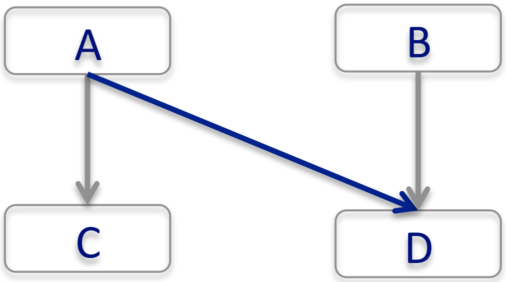
- Operations represented in Static Single Assignment Form
  - They discuss **SSA** as a graph; it can just as easily be viewed as a linear form
  - Each node in the graph represents an **SSA** name (or *value*)
    - Node has an operation & operand (edges to other nodes)
  - A set of nodes has an implicit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

Representation of calls simplifies method inlining

# Swift Compiler’s IR

## 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 <u>all critical edges</u> 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

No *deliberate* **SSA** edges between memory ops to represent order.

# 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

| Interprocedural Analyses | Interprocedural Opts | Machine Dependent Opts |
|---|---|---|
| Alias Analysis | Bound check removal | Lower **IR** |
| Class Hierarchy Analysis | Branch removal | Peephole optimizations |
| Escape Analysis | Constant propagation | Sign-extension elimination |
| Field Analysis | Dead code elimination | Trace scheduling |
| Type Propagation | Global **CSE** | Register allocation |
| Interprocedural Opts | Global code motion | Block layout |
| Method resolution | Loop peeling | Final code generation |
| Method inlining | Null check removal | |
| Method splitting | Peephole optimization | |
| Object inlining | Strength reduction | |
| Stack allocation | Type test elimination | |
| Synchronization removal | | |

# Experimental Evaluation

## Platform

High speed for 2000

- 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**

Their **JVM** plays some cute tricks, too.

## 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

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

Table 1 from the paper

# Experimental Evaluation

| | Optimizations | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *inl* | *cha* | *fld* | *objinl* | *split* | *stk* | *sync* | *sr* | *cse* | *gcm* | *peel* | *ckelim* | *selim* | *br* |
| `compress` | 1.16 | 1.20 | 1.16 | | | | | 1.09 | 1.06 | | | | 1.04 | |
| `jess` | 1.07 | 1.09 | | | | | | | 1.04 | | 1.03 | 1.04 | | |
| `cst` | 1.08 | 1.04 | | | | | 1.05 | | 1.07 | | | | | |
| `db` | 1.05 | 1.26 | 1.04 | 1.03 | 1.03 | 1.04 | | | | | 1.03 | | | |
| `si` | 1.27 | 1.14 | 1.05 | 1.04 | 1.06 | 1.16 | | | 1.12 | | | | 1.04 | 1.09 |
| `javac` | 1.09 | 1.09 | | | | | | | | | | | | |
| `mpeg` | 1.07 | | 1.13 | | | | | 1.05 | 1.35 | | | | | |
| `richards` | 1.40 | 1.76 | | | | | | | | | | | | 1.11 |
| `mtrt` | 1.57 | 2.68 | 1.27 | 1.16 | | 1.13 | | 1.09 | 1.06 | | | | | |
| `jack` | | 1.05 | | | | | | | | | | | | |
| `tsgp` | 1.03 | 1.05 | | | | | | 1.12 | 1.05 | | 1.05 | | | |
| `jlex` | 1.22 | 1.19 | | 1.15 | 1.18 | | 1.15 | | | | | | | |

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)