Fast, Effective Code Generation in a Just-In-Time (JIT) Java Compiler

Authors
Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh and James M. Stichnoth
Intel Corporation

Presenter
Debashis Panigrahi

Outline

- Java Compilation
- Just-In-Time Compiler
- Code Generation
  - Lazy Code Selection
  - Common Sub-expression Elimination
  - Register Allocation
  - Example
- Experimental Result
- Conclusion
- State of the Art
Java Compilation

- Java Program
  - Cross Compiler
  - Java Compiler
  - Machine Binary
    - Interpreter
    - JIT Compiler
  - Java bytecode
- OS
- CPU

- Java OS
- Java CPU

JIT Compiler

- Interface between JVM and JIT Compiler
  - JIT Compiler compiles bytecode to native whenever instructed by VM

- APIs defined to control JIT Compiler from VM

- Issues
  - Compilation Time: Lightweight and Effective Optimization
  - Mapping of stack operands and Local variables
  - Handling of Exceptions
  - Support for Garbage Collection
Compiler Passes

Prepass
  → Collects information about
     - Depth of operand Stack at basic block
     - Static reference count of local variable
  → Maps physical variables to local registers

Global register allocation
→ Lazy Code Selection

Code generation
→ Copies code and data section to final memory location

Code emission
→ Fixes up relocation

Code Generation : Lazy Code Selection

Goal
- Keep Java operand stack values in scratch registers
- Take advantage of IA32 addressing modes by folding loads of operands to compute instructions that use them

Data Structure
- Mimic Stack: simulates Java run-time operand stack at JIT Time
- Operand Class: useful for folding operands
Code Generation: Lazy Code Selection...contd

Code Selection

Input: Byte Code B
1. Pop operand from *Mimic Stack*
2. Try to fold operands to compute inst.
3. If operand O cannot be folded then
   select inst. that loads O into a scratch reg R
4. Push the result of computation to *Mimic Stack*

- At Call Site: generates spills for operands live across the call site
- What if Java operand stack is not empty at entry or exit of basic block?
  - At exit of basic block: *mimic stack values are spilled to stack frame*
  - At branch target: *mimic stack locations are initialized*

Simple Optimizations

- Strength Reduction
  - Converts compare followed by branch instructions to one compare and branch instruction
- Elimination of redundant load after store

Retargetable to RISC architectures
Code Generation: Common Subexpression Elimination

- Java bytecode used for representing expressions
  
  Example: “X+Y” => [iLoad_1, iLoad_2, iAdd] => [1b1c60]
  
  Expressions represented by “expression tag” = <offset,length>
  
  Scratch registers are annotated with the tag of expression

- Code selectors looks ahead to find whether any scratch register contains the result of the expression it is going to compute

- Maintenance of expression tags in a register R
  - Expression in R is killed
    - Instructions that modify R: Call Site, Register Allocator
    - Assignments that modify value of R: assignment to operand etc.

Code Generation: Common Subexpression Elimination...contd

- Limitations
  - Cannot re-associate expressions
    - For example: “x + y” and “y + x” are treated as different expr.
  - Cannot detect expressions with same value
    - For example: “x = w; x+y” and “w+y”
  - Can only represent contiguous byte code expression
    - No bubble or gap is allowed
Code Generation : Register Allocation

- **IA32 Register Architecture**
  - 3 caller-saved registers (eax, ecx, and edx) : Local Register
  - 4 callee-saved registers (ebx, ebp, esi, and edi) : Global Register

- **Local Register Allocation**
  - Register used for expression evaluation by code selector
  - Heuristic: Allocates a register that is least recently used

- **Global Register Allocation**
  - **Heuristic 1**: Allocate 4 callee-saved registers to the 4 variables with highest static reference counts
    - **Limitation**: Does not allow two variables with non-overlapping live ranges to share a register
    - **Complexity**: $O(B)$ where $B =$ Number of byte codes in the method

Code Generation : Register Allocation...contd

- **Heuristic 2**: Priority Based Register Allocation [F.C.Chow et al]
  - without use of an interference graph

**Algorithm**

For each variable, perform backward DFS through Flow Graph terminating at definition of the variable

- Keeps track of unavailable callee-saved register
- If register R is available to all visited basic blocks, then allocate R to the variable

**Complexity**: $O(B+NV)$  
$N =$ No of basic blocks  
$V =$ No of variables
Code Generation : Miscellaneous

- Array bounds check elimination
  - Java requires array access checking at runtime
  - Eliminates bound checks for constant indices
    - Example: If A[7] is checked, there is no need to check A[5]

- Out-of-line exception throws

```
 cmp [eax+offset(length)], ecx
 ja OK
 ........ ; throw an exception
 OK:    ; access the array
         ; element

 cmp [eax+offset(length)], ecx
 jbe NOTOK
 ........ ; access the array
         ; element
 NOTOK: ; throw an exception
```

Code Generation : Example
Experimental Result

- **Comparison of runtime performance**

<table>
<thead>
<tr>
<th></th>
<th>SDIC 3.0</th>
<th>SDIC 2.5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Microsoft</td>
<td>Total run-time</td>
</tr>
<tr>
<td>Backprop</td>
<td>20.61</td>
<td>44.97</td>
</tr>
<tr>
<td>Computer</td>
<td>3.35</td>
<td>5.40</td>
</tr>
<tr>
<td>Java Cup</td>
<td>1.39</td>
<td>1.70</td>
</tr>
<tr>
<td>Tweet</td>
<td>12.28</td>
<td>12.65</td>
</tr>
<tr>
<td>Loom</td>
<td>90.75</td>
<td>90.31</td>
</tr>
</tbody>
</table>

- **Comparison of optimization**

- **Conclusion**

- Presents a JIT Compiler for IA32 architecture
- Proposed “Lazy Code Selection” : a method to generate code
- Lightweight implementation of standard optimizations
  - Quality of output vs. Compilation Time trade-off
- Discussions
  - Pre-allocation of register for local variables may not generate optimized code
  - Spilling of all variables in mimic stack at join points can be avoided
State of the Art

- Kaffe (www.kaffe.org)
  - Many loads and stores in the translated code
- Vtune: IA32 Architecture
- CACAO: Alpha Platform
  - Converts bytecode to intermediate format
  - Preallocation of register followed by one pass lazy code selection
- Latte: Sparc Platform developed at Seoul National Univ
  - Two pass code generation using intermediate format
- Microsoft JIT
- Sun JIT
- OpenJIT: jointly developed by Fujitsu Labs and TIT
  - Written in Java with a flexible framework
  - Have notion of "compilets": applets for different kinds of optimizations