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

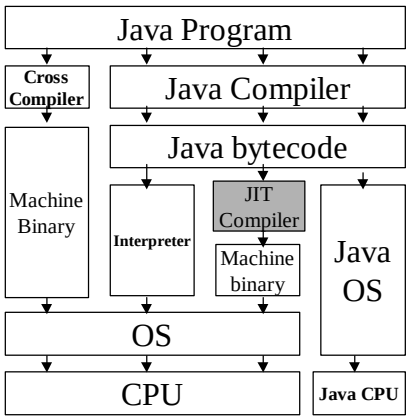
Authors
Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh,
Vishesh M. Parikh and James M. Stichnoth
Intel Corporation

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

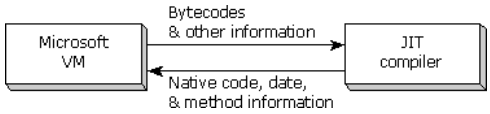
Java Compilation



JIT Compiler

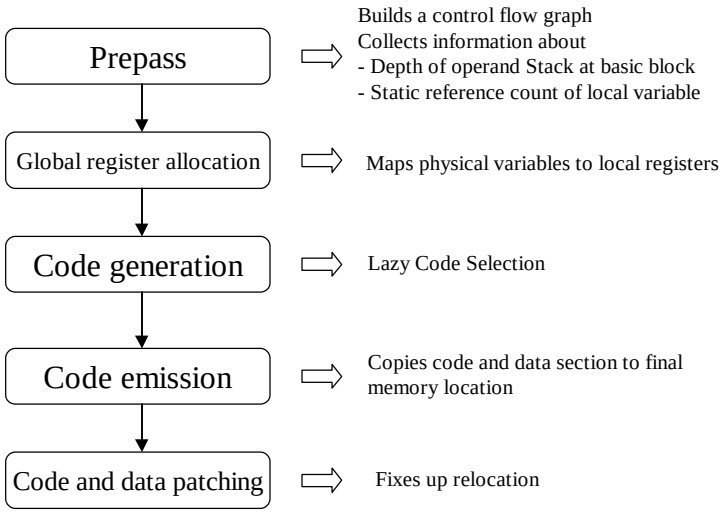
- Interface between JVM and JIT Compiler

  JIT Compiler compiles bytecode to native whenever instructed by VM



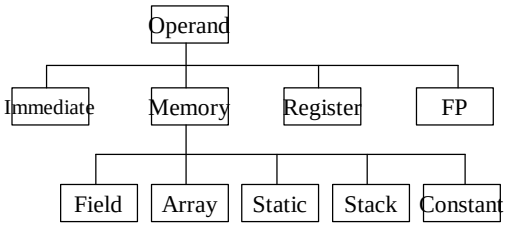
- APIs defined to control JIT Compiler from VM
- Issues
  - Compilation Time : Lightweight and Effective Optimization
  - Mapping of stack operands and Local variables
  - Handling of Exceptions
  - Support for Garbage Collection

Compiler Passes



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 can not 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*

Code Generation : Lazy Code Selection….contd

- Simple Optimizations
  - Strength Reduction
  - Coverts 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

<u>Algorithm</u>

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

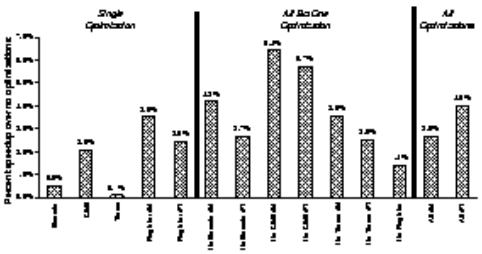
	IA32 native code	eax	ecx	edx
483: aload_0				
484: getfield #34				
487: getfield #45	mov ecx, [ebp+04h]	—	**[483,4]**	—
490: getfield #44	mov edx, [ecx+190h]	—	[483,4]	**[483,7]**
493: lload 18				
495: l2i				
496: aaload	mov eax, [edx+18h]	**[483,10]**	[483,4]	[483,7]
	mov ecx, [esp+128h]	[483,10]	**[493,3]**	[483,7]
	cmp [eax+04h], ecx			
	jbe _throw			
497: getfield #16	mov edx, [eax+ecx*4+08h]	[483,10]	[493,3]	**[483,14]**
500: i2i				
501: istore 5	mov eax, [edx+04h]	**[483,18]**	[493,3]	[483,14]
	mov [esp+160h], eax			
503: aload_0				
504: getfield #34				
507: getfield #45				
510: getfield #44				
513: lload 18				
515: l2i				
516: aaload				
517: getfield #49				
520: i2l	mov ecx, [edx+0Ch]			
	mov edx, ecx			
	sar edx, 1Fh			
521: lstore 20	mov [esp+124h], edx			
	mov [esp+120h], ecx			

Experimental Result

- Comparison of runtime performance

	SDK 1.5.1				SDK 2.0			
	Microsoft	Intel (no opts)	Intel (all opts, simple allocator	Intel (all opts, priority allocator	Microsoft	Intel (no opts)	Intel (all opts, simple allocator	Intel (all opts, priority allocator
Backprop	40.94	74.52	41.51	41.56	31.78	73.01	41.72	41.39
Compress	3.33	3.46	2.72	2.72	3.00	3.61	2.94	2.96
Java Cup	1.29	1.20	1.20	1.21	0.94	1.01	1.00	1.08
Go	14.62	15.33	11.14	11.33	9.35	11.66	8.05	8.06
Jpeg	12.03	12.81	11.84	11.75	6.61	8.04	7.12	7.12
Lisp	39.73	44.10	40.52	40.84	14.42	18.72	18.23	17.97

- 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 store 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 kind of optimizations