# One-pass Code Generation in V8

Kevin Millikin
Google

**As I tell my compiler students now, there is a fine line between “optimization” and “not  being stupid.”**

**--** *R. Kent Dybvig, The Development of Chez Scheme, ICFP 2006*

# V8 Overview

- V8: JavaScript engine used in Google Chrome, Android, node.js, etc.
- Two different code generator back ends
  - "Classic" has lots of JS-specific optimizations
  - "New" quickly produces compact code
- Both generate code in one pass from the AST
- No intermediate language!  No interpreter!

# Simple One-Pass Code Generation

- Recursively traverse the AST
- Generate code for each node
- In terms of the code for its child subtrees
- Lots of examples will follow

# Let's Use a Simple Execution Model

- Compile as if for a stack machine
- Use the call stack to store intermediate values
- Local variables can also be found in the call stack

# Example: Compiling Addition

Emit(AddExpr e) =
  { Emit(e.left) }
  { Emit(e.right) }
 **pop ebx**
 **pop eax**
 **add eax, ebx**
 **push eax**

# Example: Variables and Literals

Emit(VarRef e) =
**push [ebp**+{ e.offset }**]**

Emit(IntLit e) =
**push** { e.value }

# Example: Assignments

Emit(VarAssign e) =
{ Emit(e.right) }
**mov eax, [esp]**
**mov [ebp+{ e.var.offset }], eax**

Emit(ExprStmt s) =
{ Emit(s.expr) }
**pop eax**

# Compilation of "i=j+1"

**push [ebp+**{ j.offset }**]**
**push** { 1.value }
**pop ebx**
**pop eax**
**add eax, ebx**
**push eax**
**mov eax, [esp]**
**mov [ebp+**{ i.offset }**], eax**
**pop eax**

# We're Being Stupid

- Locally there is some bad code
- Redundant or unnecessary moves
- Extra memory traffic

# One Solution: Peephole Optimization

- Scan a small window of instructions at a time
- Pattern match on known bad code
- Optimize code by local rewriting

# Peephole Optimization Example

```
push [ebp+{ j.offset }]
push { 1.value }
pop ebx
pop eax
add eax, ebx
push eax
mov eax, [esp]
mov [ebp+{ i.offset }],eax
pop eax
```

```
push [ebp+{ j.offset }]
mov ebx, { 1.value }
pop eax
add eax, ebx
push eax
mov [ebp+{ i.offset }],eax
pop eax
```

# Drawbacks

- Handles fixed, known patterns
- Easy to inadvertently defeat it
- Can be difficult to implement in one pass
- The two-pass approach has high overhead

We had this in V8 but took it out

# Another Solution: Top-of-stack Caching

- Execution model is still a stack machine
- The top element of the stack is kept in a register
- "Pushing" and "popping" preserve the cached TOS
- Can avoid some unnecessary memory traffic

# Pushing and Popping

Push(Operand o) =

**push eax**

**mov eax, o**

Pop(Operand o) =

**mov o, eax**

**pop eax**

Drop() =

**pop eax**

# Addition Revisited

```
Emit(AddExpr e) =
   { Emit(e.left) }
   { Emit(e.right) }
  pop ebx
  add eax, ebx

Emit(VarRef e) =
  { Push([ebp+e.offset]) }

Emit(IntLit e) =
  { Push(e.value) }
```

# Addition Revisited, Continued

```
Emit(VarAssign e) =
    { Emit(e.right) }
  mov [ebp+{ e.var.offset }], eax

Emit(ExprStmt s) =
    { Emit(s.expr) }
    { Drop() }
```

# Putting It Together: "i=j+1"

```
push eax
mov eax, [ebp+{ j.offset }]
push eax
mov eax, { 1.value }
pop ebx
add eax, ebx
mov [ebp+{ i.offset }], eax
pop eax
```

# Compare (TOS Caching - Peephole)

```
push eax
push [ebp+{ j.offset }]
mov eax, { 1.value }
pop ebx
add eax, ebx
mov [ebp+{ i.offset }], eax
pop eax
```

```
push [ebp+{ j.offset }]
mov ebx, { 1.value }
pop eax
add eax, ebx
push eax
mov [ebp+{ i.offset }],eax
pop eax
```

# Drawbacks

- Some values needlessly cycled through cache
- Still needs some peephole optimization
- Have to manage two states (cached/not cached)

We also had this in V8 but took it out

# Our Solution: DDCG

- Why peephole optimization works: it can look at both sides of the boundary between AST nodes
- Why TOS caching works: it optimistically assumes every subtree is a rightmost one
- Can we do better? Destination-Driven Code Generation (DDCG)
- Parent nodes tell their children where they want values

# Example: Addition Again

```
Emit(AddExpr e, Dest d) =
   { Emit(e.left, STACK) }
   { Emit(e.right, ACCUMULATOR) }
 pop ebx
 add eax, ebx
   { Plug(d, eax) }
```

# Example Continued: Leaf Nodes

Emit(VarRef e, Dest d) =
    { Plug(d, [ebp+e.offset] }

Emit(IntLit e, Dest d) =
    { Plug(d, e.value) }

# Example Continued: Assignment

Emit(VarAssign e, Dest d) =
  { Emit(e.right, ACCUMULATOR) }
 **mov [ebp+**{ e.var.offset }**], eax**
  { Plug(d, eax) }

Emit(ExprStmt s) =
  { Emit(s.expr, NOWHERE) }

# Plugging is the Key (and easy)

Plug(STACK, eax) =
 **push eax**

Plug(ACCUMULATOR, eax) =
  // Nothing to do.

Plug(NOWHERE, eax) =
  // Nothing to do.

# More Plugging

Plug(STACK, Memory m) =
  **push m**

Plug(ACCUMULATOR, Memory m) =
  **mov eax, m**

Plug(NOWHERE, Memory m) =
    // Nothing to do.

# More Plugging

Plug(STACK, Literal L) =
  **push L**

Plug(ACCUMULATOR, Literal L) =
  **mov eax, L**

Plug(NOWHERE, Literal L) =
    // Nothing to do.

# Putting It Together: "i=j+1"

{ Plug(STACK, [ebp+j.offset] }
{ Plug(ACCUMULATOR, 1.value) }
**pop ebx**
**add eax, ebx**
{ Plug(ACCUMULATOR, eax) }
**mov [ebp+**{ i.offset }**], eax**
{ Plug(NOWHERE, eax) }

# After Plugging

```
push [ebp+{ j.offset }]
mov eax, { 1.value }
pop ebx
add eax, ebx
mov [ebp+{ i.offset }], eax
```

# Compare (DDCG - TOS Caching)

```
push [ebp+{ j.offset }]
mov eax, { 1.value }
pop ebx
add eax, ebx
mov [ebp+{ i.offset }], eax
```

```
push eax
push [ebp+{ j.offset }]
mov eax, { 1.value }
pop ebx
add eax, ebx
mov [ebp+{ i.offset }], eax
pop eax
```

# Compare (DDCG - Peephole)

```
push [ebp+{ j.offset }]
mov eax, { 1.value }
pop ebx
add eax, ebx
mov [ebp+{ i.offset }], eax
```

```
push [ebp+{ j.offset }]
mov ebx, { 1.value }
pop eax
add eax, ebx
push eax
mov [ebp+{ i.offset }],eax
pop eax
```

# Other Expressions: Boolean Values

```
Emit(LessThanExpr e, Dest d) =
    { Emit(e.left, STACK) }
    { Emit(e.right, ACCUMULATOR) }
  pop ebx
  cmp ebx, eax
  jnl if_false
    { Plug(d, true_value) }
  jmp done
if_false:
    { Plug(d, false_value) }
done:
```

# Compilation of Control Flow

```
Emit(IfStmt s) =
    { Emit(s.cond, ACCUMULATOR) }
  cmp eax, true_value
  jne else
    { Emit(s.then) }
  jmp exit
else:
    { Emit(s.else) }
exit:
```

# Putting This Together

```
  cmp ebx, eax
  jnl if_false
  mov eax, true_value
  jmp done
if_false:
  mov eax, false_value
done:
  cmp eax, true_value
  jne else
  { Emit(s.then) }
  jmp exit
else:
  { Emit(s.else) }
exit:
```

# Another Problem

- We're materializing true or false based on a branch, then testing them in order to branch
- Hard to eliminate with peephole optimization
- The moral equivalent of TOS caching is nasty
- DDCG to the rescue!

# Control Destinations

- In addition to a data destination, pass a control destination down to subtrees
- Control destinations can be the next instruction or a pair of labels (true and false targets)
- Plugging a value into a destination also considers the control destination

# Example: If Statements

Emit(IfStmt s) =
  { Emit(s.cond, NOWHERE, (then, else)) }
**then:**
  { Emit(s.then) }
 **jmp exit**
**else:**
  { Emit(s.else) }
**exit:**

# Example: Comparisons

Emit(LessThanExpr e, DDest d, CDest c) =
 { Emit(e.left, STACK) }
 { Emit(e.right, ACCUMULATOR) }

```
pop ebx
cmp ebx, eax
```

 { Plug(d, c, lt) }

# Plugging Into Control Destinations

Plug(NOWHERE, (true, false), eax) =

```
  cmp eax, false_value
  jeq false
  jmp true
```

Plug(ACCUMULATOR, (true, false), cond) =

```
  j[cond] materialize_true
  mov eax, false_value
  jmp false
materialize_true:
  mov eax, true_value
  jmp true
```

# Plugging Into Control Destinations

Plug(NOWHERE, (true, false), cond) =

```
j[cond] true
jmp false
```

# Control Flow Revisited

```
  cmp ebx, eax
  jlt then
  jmp else
then:
  { Emit(s.then) }
  jmp exit
else:
  { Emit(s.else) }
exit:
```

# Still Not Ideal

- We will have jumps to the next instruction:

```
  j[cond] other
  jmp next
next:
```

- Or else branches around jumps:

```
  j[cond] next
  jmp other
next:
```

- Solution is a third label which is the fall through

# Compilation of If, again

```
Emit(IfStmt s) =
    { Emit(s.cond, NOWHERE, (then, else, then)) }
then:
    { Emit(s.then) }
  jmp exit
else:
  { Emit(s.else) }
exit:
```

# Tweak Plugging

Plug(NOWHERE, (true, false, true), cond) =
**j[!cond] false**

Plug(NOWHERE, (true, false, false), cond) =
**j[cond] true**

Plug(NOWHERE, (true, false, _), cond) =
**j[cond] true**
**jmp false**

# Control Flow, finally

```
  cmp ebx, eax
  jnl else
then:
    { Emit(s.then) }
  jmp exit
else:
    { Emit(s.else) }
exit:
```

# Advantages of DDCG

- Can eliminate most redundant or unnecessary moves
- Can avoid unnecessary materialization/testing of values
- Can avoid most silly jumps and branches
- Operates efficiently in one pass
- Amazingly simple to implement!

Bugs in the compiler are NOT fun.