Native Code Generation

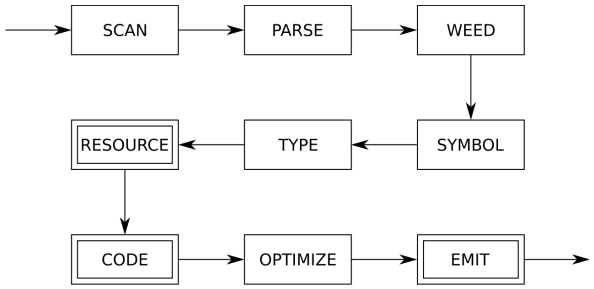
COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 8:30-9:30, TR 1080

http://www.cs.mcgill.ca/~cs520/2019/



Compiley “Mompiler” McCompilerface

Announcements (Wednesday, March 20th)

Friday's class

- Tutorial on Go codegen;
- Recommended to get tips and tricks for the codegen part of your project.

Milestones

- Milestone 2 nearly finished grading
- Milestone 3 due: Wednesday, March 27th 11:59 PM
- Milestone 4 due: Wednesday, April 10th 11:59 PM
- Final report due: Wednesday, April 10th 11:59 PM
- Group meeting: Week of April 8th (you may request an extension until the week of April 15th)
- Peephole due: Friday, April 12th 11:59 PM
- Final exam: Thursday, April 18th 2:00 PM

Executing JOOS Code

The JOOS compiler translates a subset of Java to bytecode, but bytecode cannot be executed on any machine directly. Execution occurs thanks to either

1. An interpreter;
2. An Ahead-Of-Time (AOT) compiler; or
3. A Just-In-Time (JIT) compiler.

In all cases, bytecode must be implicitly or explicitly translated into native code suitable for the host architecture before execution.

Interpreters

Execute bytecode one instruction at a time, by simulating the progression on a virtual machine

- “Easy” to implement;
- Can be very portable since they are target independent; but
- Suffer an inherent inefficiency.

Interpreters

As we saw in an earlier lecture, a simple interpreter for Java bytecode is quite easy to implement

```
pc = code.start;
while(true)
{
   npc = pc + instruction_length(code[pc]);
   switch (opcode(code[pc]))
   {
      case ILOAD_1:
         push(local[1]);
         break;
      case ILOAD:
         push(local[code[pc+1]]);
         break;
      case ISTORE:
         t = pop();
         local[code[pc+1]] = t;
         break;
      case IADD:
         t1 = pop(); t2 = pop();
         push(t1 + t2);
         break;
      case IFEQ:
         t = pop();
         if (t == 0) npc = code[pc+1];
         break;
      ...
   }
   pc = npc;
}
```

Ahead-of-Time Compilers

- Typically run on the developer's machine;
- Translate the low-level intermediate form into native code;
- Create object files to be linked and executed.

This is not so useful for Java and JOOS

- Method code is fetched as it is needed;
- From across the internet; and
- From multiple hosts with *different native code sets.*

Just-in-Time Compilers

Just-in-time (JIT) compilers are an alternative to interpreters and AOT compilerw which

- Merge interpreters with traditional compilation techniques;
- Have the overall structure of an interpreter; but
- Method code is handled differently.

When a method is invoked for the first time

- The bytecode is fetched;
- It is translated into native code; and
- Control is given to the newly generated native code.

When a method is invoked subsequently

- Control is simply given to the previously generated native code.

Just-in-Time Compilers

Efficiency

For a JIT compiler to be worthwhile

- It must be *fast*, because the compilation occurs at runtime (Just-In-Time is really Just-Too-Late);
- It may concurrently interpret and compile a method (Better-Late-Than-Never); and
- It may have several levels of optimization, and recompile long-running methods.

Cutting corners

Since we require high performance

- It does not necessarily generate optimized code;
- It does not necessarily compile every instruction into native code, but relies on the runtime library for complex instructions; and
- It need not necessarily compile every method;

Generating Native Code

When generating native code, there are 4 important problems to solve

- **Instruction selection**
  Choose the correct instructions based on the native code instruction set;
- **Memory modelling**
  Decide where to store variables and how to allocate registers;
- **Method calling**
  Determine calling conventions; and
- **Branch handling**
  Allocate branch targets.

Compiling JVM Bytecode to VirtualRISC

In this class we focus on generating VirtualRISC from JVM bytecode. In a JIT compiler this requires

- Mapping the Java local stack into registers and memory;
- Instruction selection on the fly;
- Branch target allocation on the fly.

This is successfully done in the Kaffe system.

Mapping Locals/Stack to the Frame

The first task before generating native code is to place variables in memory

```
name      offset   location   register

a         1        [fp-4]

stack     0                   R1
stack     1                   R2

scratch   0                   R3
```

We have a choice of storing variables in memory (RAM) locations, and/or in registers.

Stack memory

- In theory, the stack frame can become arbitrarily large, so there will always be enough space;
- In practice, the frame is limited by the stack limit (`ulimit -a`).

Registers

- Limited number depending on the hardware;
- Usually much less than the number of program variables; and
- Requires spilling if there are not enough registers.

Register Allocation

Problem: Find the mapping scheme which keeps as many variables in registers as possible

- Requires program analysis to understand loads/stores and operations;
- In particular, the ranges over which variables are used.

Liveness analysis

A variable is *live* if its value may be read at some point in the future

- Exact liveness is undecidable; but
- Using static analysis, we can determine conservative sets of variables which are live for each program point;
- The more precise the liveness information, the better the register allocation.

Liveness Analysis

A detailed view at liveness analysis is presented in COMP 621 (some material is also available from past COMP 520 slides - `https://www.cs.mcgill.ca/~cs520/2015/slides/staticanalysis.pdf`)

For this class, we will focus on small, human-solvable problems.

Examples

For the following programs, determine the set of variables which are live at each program point.

```
a := 0
b := 1
c := 0
c := a + b
d := 5
e := b + c + d
return e
```

```
a := 0
b := 1
if a > b {
    c := 0
} else {
    c := a
}
d := b + c
return d
```

```
a := 0
b := 1
while a > b {
    a := a - 1
}
c := a
return d
```

Interference Graph

An *interference graph* represents liveness relationships between program variables.

- One node for each program variable; and
- Edges between variables whose live ranges overlap (i.e. live at the same time).

Some approaches also include a k-clique for the registers, but it isn't always necessary.

Construction

We can construct the interference graph using the results of live variable analysis.

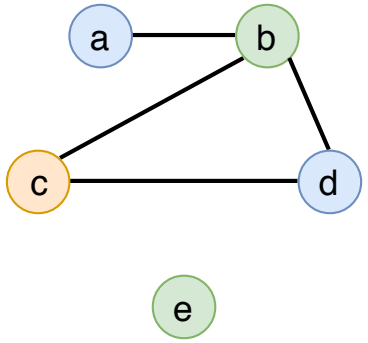
1. For each program variable n, add one node n to the graph
2. For each program point p, add an edge (a, b) to the graph if variables a and b are both live at point p

Graph Colouring

The last step in register allocation performs the mapping using the interference graph.

If the system has n registers, we want to find if the graph is n-colourable, and what the colouring would be. As register allocation reduces to graph colouring, and is therefore NP-complete.

Example



In practice, a heuristic approach to graph colouring is used to approximate the solution.

Live Range Splitting

Variables may be redefined throughout the program, essentially acting as separate variables. In the following example, a naive approach requires 3 registers, yet only 2 are required.

```
a := 0
b := 1
c := a + b
a := b
d := a + c
return d
```

We can *split* the live range and represent the two definitions of `a` as separate variables `a0` and `a1`. This allows us to obtain a 2-colouring of the interference graph.

Announcements (Monday, March 25th)

Milestones

- Milestone 2 finished grading
- Milestone 3 due: Wednesday, March 27th 11:59 PM
- Milestone 4 due: Wednesday, April 10th 11:59 PM
- Final report due: Wednesday, April 10th 11:59 PM
- Group meeting: Week of April 8th (you may request an extension until the week of April 15th)
- Peephole due: Friday, April 12th 11:59 PM
- Final exam: Thursday, April 18th 2:00 PM

Allocation Schemes

Applying graph-based register allocation schemes directly to stack-based IR is not trivial.

In class we will instead show 3 basic schemes used for generating native code from Java bytecode which ignore the complexities of liveness and graph colouring.

- Naïve (no allocation);
- Fixed; and
- Basic block.

We will produce a mapping of both local variables and stack positions to the memory and registers.

Instruction Selection

Translating from a stack-based IR to a register-based IR is

- Non-trivial; and
- (Typically) requires generating other register-based intermediate representations.

In class we take a simple approach, simulating the stack operations to directly produce machine code. For example `iload_0` may be represented as

```
ld [fp-4], R0
st R0, [fp-8]
```

where `[fp-4]` and `[fp-8]` represent local 0 and the top of the stack respectively.

Compiling JVM Bytecode to VirtualRISC

The general algorithm for generating native code consists of

- Finding the local stack height for each bytecode;
- Determining the number of slots in frame: locals limit + stack limit + #temps;
- Mapping memory/registers;
- Emitting prologue;
- Emitting native code for each bytecode; and
- Fixing up branches.

For VirtualRSIC, we assume that input parameters arrive in registers `R0` through `Rn`, and the return value must be in `R0`

Naïve Allocation

The naïve approach assumes that registers are very volatile, where the value stored in a register from one bytecode cannot be accessed from another.

```
iload_0  // loads value into R0
ineg     // cannot assume anything about R0
```

i.e. Even though `iload_0` loads the value into `R0`, then the next instruction can make no assumption about the contents of `R0`. To transfer data, we must use memory.

Generating code

- Each local and stack location is mapped to an offset in the native frame;
- Each bytecode is translated into a series of native instructions, which
- Constantly move locations between memory and registers.

This is similar to the native code generated by a non-optimizing compiler.

Naïve Allocation Example

Input code

```
public void foo() {
   int a, b, c;

   a = 1;
   b = 13;
   c = a + b;
}
```

Procedure

- Compute frame size = 4 + 2 + 0 = 6;
- Map locals/stack to frame;
- Find stack height for each bytecode;
- Emit prologue; and
- Emit native code for each bytecode.

Generated bytecode

```
.method public foo()V
.limit locals 4
.limit stack 2
  iconst_1          ; 1
  istore_1          ; 0
  ldc 13            ; 1
  istore_2          ; 0
  iload_1           ; 1
  iload_2           ; 2
  iadd              ; 1
  istore_3          ; 0
  return            ; 0
.end method
```

Naïve Allocation Example

Assignment of frame slots

name	offset	location
a	1	[fp-32]
b	2	[fp-36]
c	3	[fp-40]
stack	0	[fp-44]
stack	1	[fp-48]

Native code generation

```
                          save sp,-136,sp
a = 1;        iconst_1    mov 1,R1
                          st R1,[fp-44]
              istore_1    ld [fp-44],R1
                          st R1,[fp-32]
b = 13;       ldc 13      mov 13,R1
                          st R1,[fp-44]
              istore_2    ld [fp-44],R1
                          st R1,[fp-36]
c = a + b;    iload_1     ld [fp-32],R1
                          st R1,[fp-44]
              iload_2     ld [fp-36],R1
                          st R1,[fp-48]
              iadd        ld [fp-48],R1
                          ld [fp-44],R2
                          add R2,R1,R1
                          st R1,[fp-44]
              istore_3    ld [fp-44],R1
                          st R1,[fp-40]
              return      restore
                          ret
```

NaÏve Allocation

Naïve allocation is not a very good allocation scheme (although it is typically used when the optimizer is turned off - always use $-O$!)

- Clear very slow;
- Many unnecessary loads and stores, which
- Are the *most* expensive operations.

We wish to replace repeated loads and stored with register operations

Improved Allocation

We wish to replace loads and stores

```
c = a + b;   iload_1    ld [fp-32],R1
                        st R1,[fp-44]
             iload_2    ld [fp-36],R1
                        st R1,[fp-48]
             iadd       ld [fp-48],R1
                        ld [fp-44],R2
                        add R2,R1,R1
                        st R1,[fp-44]
             istore_3   ld [fp-44],R1
                        st R1,[fp-40]
```

by registers operations

```
c = a + b;   iload_1    ld [fp-32],R1
             iload_2    ld [fp-36],R2
             iadd       add R1,R2,R1
             istore_3   st R1,[fp-40]
```

where `R1` and `R2` represent the stack.

Announcements (Wednesday, March 27th)

Milestones

- Milestone 3 due: Wednesday, March 27th 11:59 PM
- Milestone 4 due: Wednesday, April 10th 11:59 PM
- Final report due: Wednesday, April 10th 11:59 PM
- Group meeting: Week of April 8th (you may request an extension until the week of April 15th)
- Peephole due: Friday, April 12th 11:59 PM
- Final exam: Thursday, April 18th 2:00 PM

Fixed Register Allocation

- Assign m registers to the first m locals;
- Assign n registers to the first n stack locations;
- Assign k scratch registers; and
- Spill remaining locals and locations into memory.

Example

Given 6 registers ($m = n = k = 2$), show the register allocation table

name	offset	location	register
a	1		R1
b	2		R2
c	3	[fp-40]	
stack	0		R3
stack	1		R4
scratch	0		R5
scratch	1		R6

Fixed Register Allocation Example

Memory allocation map

name	offset	location	register
a	1		R1
b	2		R2
c	3	[fp-40]	
stack	0		R3
stack	1		R4
...			

Improved native code generation

```
                          save sp,-136,sp
a = 1;        iconst_1    mov 1,R3
              istore_1    mov R3,R1
b = 13;       ldc 13      mov 13,R3
              istore_2    mov R3,R2
c = a + b;    iload_1     mov R1,R3
              iload_2     mov R2,R4
              iadd        add R3,R4,R3
              istore_3    st R3,[fp-40]
              return      restore
                          ret
```

Fixed Register Allocation

- Registers are allocated once; and
- The allocation does not change within a method.

This works quite well if

- The architecture has a large register set;
- The stack is small most of the time; and
- The first locals are used most frequently.

Advantages

- It's simple to do the allocation; and
- No problems with different control flow paths.

Disadvantages

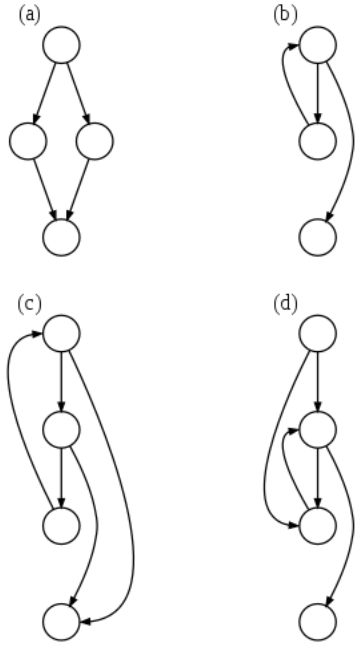
- Assumes the first locals and stack locations are most important; and
- May waste registers within a region of a method.

Basic Blocks

A *basic block* is a sequence of instructions that

- Are linear;
- Have no incoming or outgoing branches except at boundaries; and thus
- Have only one entry point (the start) and only one exit point (the end).

Basic blocks form the nodes of a *control flow graph*.



Wikimedia User: JMP EAX

Basic Block Register Allocation

- Assign frame slots to registers on demand within a basic block; and
- Update *descriptors* at each bytecode.

The descriptor maps a slot to an element of the set { ⊥, `mem`, `R`i, `mem&R`i }

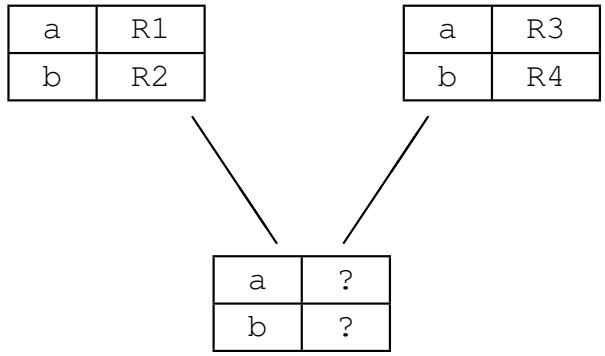
a	R2
b	mem
c	mem&R4
s_0	R1
s_1	⊥

We also maintain the inverse register map

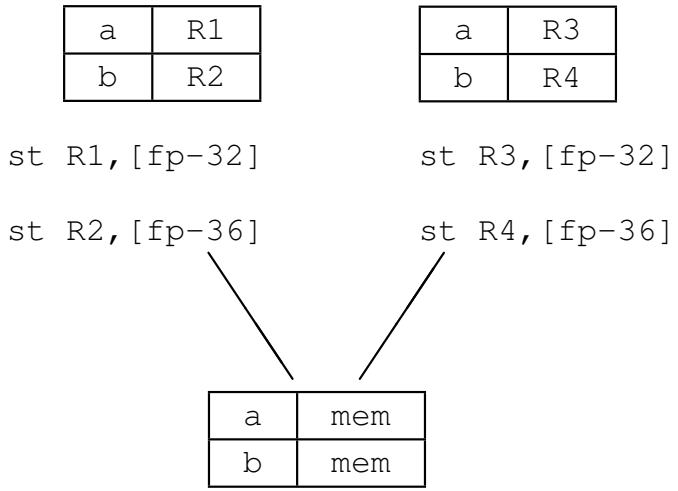
R1	s_0
R2	a
R3	⊥
R4	c
R5	⊥

Basic Block Register Allocation

In a control flow graph, divergent paths can merge at basic block boundaries (i.e. loops, if statements, etc.)



To correctly merge the paths, registers must be spilled after basic blocks



At the beginning of a basic block, all slots are therefore assumed to be in memory.

Basic Block Register Allocation Example

	`save sp,-136,sp`	R1	⊥	a	mem
		R2	⊥	b	mem
		R3	⊥	c	mem
		R4	⊥	s_0	⊥
		R5	⊥	s_1	⊥
`iconst_1`	`mov 1,R1`	R1	s_0	a	mem
		R2	⊥	b	mem
		R3	⊥	c	mem
		R4	⊥	s_0	R1
		R5	⊥	s_1	⊥
`istore_1`	`mov R1,R2`	R1	⊥	a	R2
		R2	a	b	mem
		R3	⊥	c	mem
		R4	⊥	s_0	⊥
		R5	⊥	s_1	⊥
`ldc 13`	`mov 13,R1`	R1	s_0	a	R2
		R2	a	b	mem
		R3	⊥	c	mem
		R4	⊥	s_0	R1
		R5	⊥	s_1	⊥
`istore_2`	`mov R1,R3`	R1	⊥	a	R2
		R2	a	b	R3
		R3	b	c	mem
		R4	⊥	s_0	⊥
		R5	⊥	s_1	⊥

```
iload_1      mov R2,R1
```

R1	s_0
R2	a
R3	b
R4	⊥
R5	⊥

a	R2
b	R3
c	mem
s_0	R1
s_1	⊥

```
iload_2      mov R3,R4
```

R1	s_0
R2	a
R3	b
R4	s_1
R5	⊥

a	R2
b	R3
c	mem
s_0	R1
s_1	R4

```
iadd         add R1,R4,R1
```

R1	s_0
R2	a
R3	b
R4	⊥
R5	⊥

a	R2
b	R3
c	mem
s_0	R1
s_1	⊥

```
istore_3     st R1,R4
```

R1	⊥
R2	a
R3	b
R4	c
R5	⊥

a	R2
b	R3
c	R4
s_0	⊥
s_1	⊥

```
             st R2,[fp-32]
return       st R3,[fp-36]
             st R4,[fp-40]
```

R1	⊥
R2	⊥
R3	⊥
R4	⊥
R5	⊥

a	mem
b	mem
c	mem
s_0	⊥
s_1	⊥

```
             restore
             ret
```

Basic Block vs. Fixed Register Allocation

So far, this is actually no better than the fixed scheme, in fact, it is worse (3 memory stores from spilling vs. 1 memory store). But if we add the statement

```
c = c * c + c;
```

Then the fixed scheme and basic block schemes generate

```
            Fixed                Basic block

iload_3     ld [fp-40],R3        mov R4,R1
iload_3     ld [fp-40],R4        mov R4,R5
imul        mul R3,R4,R3         mul R1,R5,R1
iload_3     ld [fp-40],R4        mov R4,R5
iadd        add R3,R4,R3         add R1,R5,R1
istore_3    st R3,[fp-40]        mov R1,R4
```

Note: When comparing both approaches we must consider the cost of spilling

Basic Block Register Allocation

- Registers are allocated on demand; and
- Slots are kept in registers within a basic block.

Advantages

- Registers are not wasted on unused slots; and
- Less spill code within a basic block.

Disadvantages

- Much more complex than the fixed register allocation scheme;
- Registers must be spilled at the end of a basic block; and
- We may spill locals that are never needed.

Further Optimizations

The schemes up until this point explicitly modelled the stack. If we remove this requirement, the code gets even faster

```
save sp,-136,sp
mov 1,R1
mov R1,R2
mov 13,R1
mov R1,R3
mov R2,R1
mov R3,R4
add R1,R4,R1
st R1,[fp-40]
restore
ret
```

```
save sp,-136,sp
mov 1,R2
mov 13,R3
add R2,R3,R1
st R1,[fp-40]
restore
ret
```

Further Optimizations

Peephole optimization

```
mov 1,R3      ⟹   mov 1,R1
mov R3,R1
```

The optimization is unsound if `R3` is used in a later instruction

```
mov 1,R3      ⟹   mov 1,R1
mov R3,R1
⋮                 ⋮
mov R3,R4         mov R3,R4
```

Control flow

Just like basic block allocation, care must also be taken when merging divergent control flow.

Such optimizations require dataflow analysis (COMP 621).

Method Invocation

Invoking methods in bytecode

- Evaluate each argument leaving results on the stack; and
- Emit `invokevirtual` instruction.

Invoking methods in native code

- Call library routine `soft_get_method_code` to perform the method lookup;
- Generate code to load arguments into registers; and
- Branch to the resolved address.

Up until now we ignored the scratch registers since all operations were performed on locals/stack locations. This is where scratch registers come into play

Method Invocation Example

Consider a method invocation

```
c = t.foo(a, b);
```

Memory map

name	offset	location	register
a	1	[fp-60]	R3
b	2	[fp-56]	R4
c	3	[fp-52]	
t	4	[fp-48]	R2
stack	0	[fp-36]	R1
stack	1	[fp-40]	R5
stack	2	[fp-44]	R6
scratch	0	[fp-32]	R7
scratch	1	[fp-28]	R8

Method Invocation Example

```
aload_4                 mov R2,R1
iload_1                 mov R3,R5
iload_2                 mov R4,R6
invokevirtual foo       // soft call to get address
                        ld [R2+4],R7
                        ld [R7+52],R8
                        // spill all registers
                        st R3,[fp-60]
                        st R4,[fp-56]
                        st R2,[fp-48]
                        st R6,[fp-44]
                        st R5,[fp-40]
                        st R1,[fp-36]
                        st R7,[fp-32]
                        st R8,[fp-28]
                        // make call
                        mov R8,R0
                        call soft_get_method_code
                        // result is in R0
                        // put args in R2, R1, and R0
                        st R0,[fp-32] // spill result
                        ld [fp-44],R2 // R2 := stack_2
                        ld [fp-40],R1 // R1 := stack_1
                        ld [fp-36],R0 // R0 := stack_0
                        ld [fp-32],R7 // reload result
                        jmp [R7] // call method
```

Method Invocation

- Long and costly; and
- The lack of dataflow analysis causes massive spills within basic blocks.

Handling Branches

Native code is generated linearly, from first instruction to last. When branches jump forward in the code

- The target address is not yet known (the code has not been generated);
- Assemblers normally handle this; but
- The JIT compiler produces binary code directly in memory.

Example

```
if (a < b)    iload_1          ld [fp-44], R1
              iload_2          ld [fp-48], R2
              if_icmpge 17     cmp R1,R2
                               bge ??
```

To generate branch targets

- Previously encountered branch targets are already known;
- Keep unresolved branches in a table; and
- Patch targets when the code is eventually generated.