

Native code generation

JOOS programs are compiled into bytecode.

This bytecode can be executed thanks to either:

- an interpreter;
- an Ahead-Of-Time (AOT) compiler; or
- a Just-In-Time (JIT) compiler.

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

Interpreters:

- are easier to implement;
- can be very portable; but
- suffer an inherent inefficiency:

```
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:

- translate the low-level intermediate form into native code;
- create all object files, which are then linked, and finally 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:

- merge interpreting with traditional compilation;
- 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.

Features of a JIT compiler:

- it must be *fast*, because the compilation occurs at run-time (Just-In-Time is really Just-Too-Late);
- it does not generate optimized code;
- it does not compile every instruction into native code, but relies on the runtime library for complex instructions;
- it need not compile every method; and
- it may concurrently interpret and compile a method (Better-Late-Than-Never).

Problems in generating native code:

- *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 into VirtualRISC:

- map the Java local stack into registers and memory;
- do instruction selection on the fly;
- allocate registers on the fly; and
- allocate branch targets on the fly.

This is successfully done in the Kaffe system.

The general algorithm:

- determine number of slots in frame:
locals limit + stack limit + #temps;
- find starts of basic blocks;
- find local stack height for each bytecode;
- emit prologue;
- emit native code for each bytecode; and
- fix up branches.

Naïve approach:

- 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.

Example:

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

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

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
```

- compute frame size = $4 + 2 + 0 = 6$;
- find stack height for each bytecode;
- emit prologue; and
- emit native code for each bytecode.

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:

```
a = 1;      iconst_1    save sp,-136,sp
                                     mov 1,R1
                                     st R1,[fp-44]
                                     ld [fp-44],R1
                                     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      return      restore
                                     ret
```

The naïve code is very slow:

- many unnecessary loads and stores, which
- are the *most* expensive operations.

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.

The *fixed* register allocation scheme:

- 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 for 6 registers ($m = n = k = 2$):

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

Improved native code generation:

```

a = 1;       iconst_1   save sp,-136,sp
              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

```

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.

Summary of fixed register allocation scheme:

- registers are allocated once; and
 - the allocation does not change within a method.
- 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.

The *basic block* register allocation scheme:

- 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 $\{\perp, \text{mem}, Ri, \text{mem}\&Ri\}$:

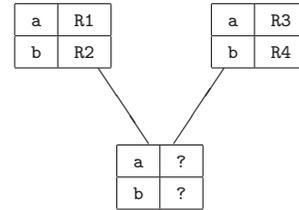
a	R2
b	mem
c	mem&R4
s_0	R1
s_1	⊥

We also maintain the inverse register map:

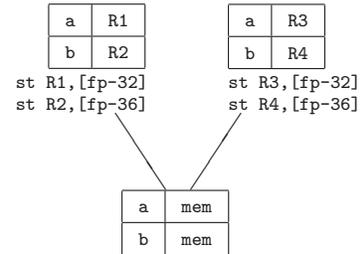
R1	s_0
R2	a
R3	⊥
R4	c
R5	⊥

At the beginning of a basic block, all slots are in memory.

Basic blocks are merged by control paths:



Registers must be spilled after basic blocks:



	save sp, -136, sp	<table border="1"><tr><td>R1</td><td>⊥</td></tr><tr><td>R2</td><td>⊥</td></tr><tr><td>R3</td><td>⊥</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table border="1"><tr><td>a</td><td>mem</td></tr><tr><td>b</td><td>mem</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>⊥</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	⊥	R2	⊥	R3	⊥	R4	⊥	R5	⊥	a	mem	b	mem	c	mem	s_0	⊥	s_1	⊥
R1	⊥																					
R2	⊥																					
R3	⊥																					
R4	⊥																					
R5	⊥																					
a	mem																					
b	mem																					
c	mem																					
s_0	⊥																					
s_1	⊥																					
iconst.1	mov 1, R1	<table border="1"><tr><td>R1</td><td>s_0</td></tr><tr><td>R2</td><td>⊥</td></tr><tr><td>R3</td><td>⊥</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table border="1"><tr><td>a</td><td>mem</td></tr><tr><td>b</td><td>mem</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>R1</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	s_0	R2	⊥	R3	⊥	R4	⊥	R5	⊥	a	mem	b	mem	c	mem	s_0	R1	s_1	⊥
R1	s_0																					
R2	⊥																					
R3	⊥																					
R4	⊥																					
R5	⊥																					
a	mem																					
b	mem																					
c	mem																					
s_0	R1																					
s_1	⊥																					
istore.1	mov R1, R2	<table border="1"><tr><td>R1</td><td>⊥</td></tr><tr><td>R2</td><td>a</td></tr><tr><td>R3</td><td>⊥</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table border="1"><tr><td>a</td><td>R2</td></tr><tr><td>b</td><td>mem</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>⊥</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	⊥	R2	a	R3	⊥	R4	⊥	R5	⊥	a	R2	b	mem	c	mem	s_0	⊥	s_1	⊥
R1	⊥																					
R2	a																					
R3	⊥																					
R4	⊥																					
R5	⊥																					
a	R2																					
b	mem																					
c	mem																					
s_0	⊥																					
s_1	⊥																					
ldc 13	mov 13, R1	<table border="1"><tr><td>R1</td><td>s_0</td></tr><tr><td>R2</td><td>a</td></tr><tr><td>R3</td><td>⊥</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table border="1"><tr><td>a</td><td>R2</td></tr><tr><td>b</td><td>mem</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>R1</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	s_0	R2	a	R3	⊥	R4	⊥	R5	⊥	a	R2	b	mem	c	mem	s_0	R1	s_1	⊥
R1	s_0																					
R2	a																					
R3	⊥																					
R4	⊥																					
R5	⊥																					
a	R2																					
b	mem																					
c	mem																					
s_0	R1																					
s_1	⊥																					
istore.2	mov R1, R3	<table border="1"><tr><td>R1</td><td>⊥</td></tr><tr><td>R2</td><td>a</td></tr><tr><td>R3</td><td>b</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table border="1"><tr><td>a</td><td>R2</td></tr><tr><td>b</td><td>R3</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>⊥</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	⊥	R2	a	R3	b	R4	⊥	R5	⊥	a	R2	b	R3	c	mem	s_0	⊥	s_1	⊥
R1	⊥																					
R2	a																					
R3	b																					
R4	⊥																					
R5	⊥																					
a	R2																					
b	R3																					
c	mem																					
s_0	⊥																					
s_1	⊥																					

iload.1	mov R2, R1	<table border="1"><tr><td>R1</td><td>s_0</td></tr><tr><td>R2</td><td>a</td></tr><tr><td>R3</td><td>b</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table border="1"><tr><td>a</td><td>R2</td></tr><tr><td>b</td><td>R3</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>R1</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	s_0	R2	a	R3	b	R4	⊥	R5	⊥	a	R2	b	R3	c	mem	s_0	R1	s_1	⊥
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	<table border="1"><tr><td>R1</td><td>s_0</td></tr><tr><td>R2</td><td>a</td></tr><tr><td>R3</td><td>b</td></tr><tr><td>R4</td><td>s_1</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table border="1"><tr><td>a</td><td>R2</td></tr><tr><td>b</td><td>R3</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>R1</td></tr><tr><td>s_1</td><td>R4</td></tr></table>	R1	s_0	R2	a	R3	b	R4	s_1	R5	⊥	a	R2	b	R3	c	mem	s_0	R1	s_1	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	<table border="1"><tr><td>R1</td><td>s_0</td></tr><tr><td>R2</td><td>a</td></tr><tr><td>R3</td><td>b</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table border="1"><tr><td>a</td><td>R2</td></tr><tr><td>b</td><td>R3</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>R1</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	s_0	R2	a	R3	b	R4	⊥	R5	⊥	a	R2	b	R3	c	mem	s_0	R1	s_1	⊥
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	<table border="1"><tr><td>R1</td><td>⊥</td></tr><tr><td>R2</td><td>a</td></tr><tr><td>R3</td><td>b</td></tr><tr><td>R4</td><td>c</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table border="1"><tr><td>a</td><td>R2</td></tr><tr><td>b</td><td>R3</td></tr><tr><td>c</td><td>R4</td></tr><tr><td>s_0</td><td>⊥</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	⊥	R2	a	R3	b	R4	c	R5	⊥	a	R2	b	R3	c	R4	s_0	⊥	s_1	⊥
R1	⊥																					
R2	a																					
R3	b																					
R4	c																					
R5	⊥																					
a	R2																					
b	R3																					
c	R4																					
s_0	⊥																					
s_1	⊥																					
	st R2, [fp-32]	<table border="1"><tr><td>R1</td><td>⊥</td></tr><tr><td>R2</td><td>⊥</td></tr><tr><td>R3</td><td>⊥</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table border="1"><tr><td>a</td><td>mem</td></tr><tr><td>b</td><td>mem</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>⊥</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	⊥	R2	⊥	R3	⊥	R4	⊥	R5	⊥	a	mem	b	mem	c	mem	s_0	⊥	s_1	⊥
R1	⊥																					
R2	⊥																					
R3	⊥																					
R4	⊥																					
R5	⊥																					
a	mem																					
b	mem																					
c	mem																					
s_0	⊥																					
s_1	⊥																					
	st R3, [fp-36]																					
	st R4, [fp-40]																					
return	restore																					
	ret																					

So far, this is actually no better than the fixed scheme.

But if we add the statement:

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

then the fixed scheme and basic block scheme generate:

	Fixed	Basic block
iload_3	ld [fp-40],R3	mv R4, R1
dup	ld [fp-40],R4	mv R4, R5
imul	mul R3,R4,R3	mul R1, R5, R1
iload_3	ld [fp-40],R4	mv R4, R5
iadd	add R3,R4,R3	add R1, R5, R1
istore_3	st R3,[fp-40]	mv R1, R4

Summary of basic block register allocation scheme:

- 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.

We can optimize further:

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

by not explicitly modelling the stack.

Unfortunately, this cannot be done safely on the fly by a peephole optimizer.

The optimization:

```
mov 1,R3    ==>  mov 1,R1
mov R3,R1
```

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
```

Such optimizations require dataflow analysis.

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.

Consider a method invocation:

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

where the memory map is:

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

Generating native code:

```

aload_4      mov R2,R1
iload_1      mov R3,R5
iload_2      mov R4,R6
invokevirtual foo // soft call to get address
              ld R7,[R2+4]
              ld R8,[R7+52]
              // 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] // R1 := stack_1
              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
              ld R2,[fp-44] // R2 := stack_2
              ld R1,[fp-40] // R1 := stack_1
              st R0,[fp-32] // spill result
              ld R0,[fp-36] // R0 := stack_0
              ld R4,[fp-32] // reload result
              jmp [R4] // call method

```

- this is long and costly; and
- the lack of dataflow analysis causes massive spills within basic blocks.

Handling branches:

- the only problem is that the target address is not known;
- assemblers normally handle this; but
- the JIT compiler produces binary code directly in memory.

Generating native code:

```

if (a < b)   iload_1      ld R1,[fp-44]
              iload_2      ld R2,[fp-48]
              if_icmpge 17  sub R1,R2,R3
              bge ??

```

How to compute the branch targets:

- previously encountered branch targets are already known;
- keep unresolved branches in a table; and
- patch targets when the bytecode is eventually reached.