Compiler construction 2012

Lecture 3

- Arrays in JVM
- JVM and optimization.
- A first look at optimization: Peephole optimization.

Array types are reference types

Variables are pointers

A local variable of an array type contains a **pointer** to the actual array. A pointer has JVM size one word.

The actual array (including the `length` field) is stored in the heap.

Array objects must be explicitly created (of a given size). When such objects are no longer referenced to, they will be **garbage collected**.

JVM instructions do all manipulation

All the bureaucracy of computing addresses for array access, checking that indices are within bounds, etc is handled by the JVM through a collection of instructions.

We only need to learn these.

Declaring and creating arrays

Variable declaration

To declare an array variable as in

```
int[] a;
double[] b;
```

will not generate any Jasmin code.
But your compiler will need to give the variables numbers and store this info in the state.

Creating an array

To create an array, as in

```
a = new int[20];
```

gives Jasmin code to the right (if `a` has variable number 3)

Jasmin code

```
bipush 20
newarray int
astore_3
```

Of course, the net stack effect of this sequence is nil.

Loading, storing and indexing

Loading an array reference

Instructions

`aload` *n*	where `n` is a constant
`aload_`*n*	where *n*=0,1,2,3

push an array reference from a local variable onto the operand stack.
No type distinction between different types of arrays.

Storing a reference

Analogous `astore` instructions store (and pop) references from top of stack.

Loading an array element

To push an array element onto the stack:

- push the array reference;
- push the index;
- execute `iaload` (for `int` arrays) resp `daload` (for `double` arrays).

Storing an array element

To store a value as an array element:
push reference, index and value and execute `iastore`/`dastore`.

Array length and the `foreach`-loop

Array length

The instruction `arraylength` gives the length of an array. What should be on the stack before execution?

The `foreach`-loop

This is the new construct

`for (` *type var* `:` *expr* `)`
stmt

where *expr* must have type *type*[].

`javac` translates this to the code to the right.

Translated code

type[] `a =` *expr*;
```
int len = a.length;
for (int i=0; i<len; i++) {
```
var `= a[i];`
stmt
`}`

where `a`, `len` and `i` are **generated**, unique variable names.

You could build on this and translate further to `while` loop.

An example

An example

Consider the following function in Javalette extended with (one-dimensional) arrays and `foreach` loops.

```
int sum (int[] a) {
     int res = 0;
     for (int x : a)
         res = res + x;
     return res;
}
```

Generated Jasmin code could be as on the next slide.
(This is the code that `javac` produces for this function as a `static` method in Java.)

Jasmin code for the example

```
.method public static sum([I)I
.limit locals 6
.limit stack 3
   iconst_0
   istore_1
   aload_0
   astore_2
   aload_2
   arraylength
   istore_3
   iconst_0
   istore 4
 lab0:
   iload 4
   iload_3
   if_icmpge lab1
   aload_2
   iload 4
   iaload
   istore 5
   iload_1
   iload 5
   iadd
   istore_1
   iinc 4 1
   goto lab0
 lab1:
   iload_1
   ireturn
.end method
```


Optimization: a simple example

A Java class

```
public class A {

 public static int f (int x) {
     int r = 3;
     int s = r + 5;
     return s * x;
 }
}
```

Questions

- Why doesn't `javac` produce better code?
- How would you do to generate good code?

Code generated by `javac`

```
.method public static f(I)I
.limit locals 3
.limit stack 2
   iconst_3
   istore_1
   iload_1
   iconst_5
   iadd
   istore_2
   iload_2
   iload_0
   imul
   ireturn
.end method
```

Measuring Java execution time

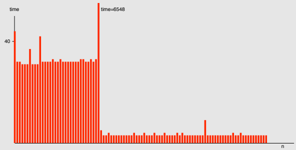
```
public class Timing {
   public static void main (String [] args) {

     for (int n = 0; n < 100; n++) {
       long start = System.nanoTime();
       sum(300);
       long stop = System.nanoTime();
       System.out.println (n+": "+(stop-start)/1000);
    }
  }

  public static int sum (int n) {
     if (n <= 1) return 1;
     else return n + sum (n-1);
  }
}
```


Running class `Timing` on Java HotSpot VM, 1

Server VM

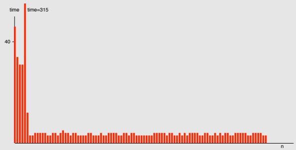


Comment

After 10000 method calls to `sum` using the interpreter, the VM decides to invest in optimising compilation of method `sum`.
This reduces execution time of future calls of `sum(300)` by 90 %.

Running class `Timing` on Java HotSpot VM, 2

Client VM



Comment

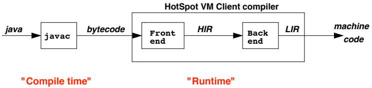
After 1500 method calls to `sum` using the interpreter, the VM decides to invest in (a less optimising) compilation.
This gives same reduction of execution time.

JIT compilation

The 80/20 rule

80 % of the time is spent running 20 % of the code.
(Some say that the correct version is the 90/10 rule.)

Conclusion: Spend the cost of compilation on the hot code only.



Java HotSpot VM

Two versions

Java HotSpot VM uses JIT compilation and comes in two versions.

- Server VM. Focuses on overall performance.
  Default on server-class machines.
- Client VM. Focuses on short startup time and small footprint. Default on smaller machines.

Core VM

Same; only compilers (from bytecode to machine code) different.

Recently, major progress in making locking more efficient.

Garbage collection strategies, heap sizes, etc can be tuned.

A surprising (?) fact

Java HotSpot VM is written in C++.

Java HotSpot client compiler

General structure

- Front end. From bytecode to HIR (High level Intermediate).

  HIR is SSA-based, control-flow graph representation of bytecode.

  Some code optimizations:

  - copy propagation.
  - common subexpression elimination.
  - constant folding.
  - inlining.

  Method call optimization (static calls instead of dynamic).
- Back end. From HIR to LIR, then to machine code.

  Simple but good register allocation (after Easter).

  Peephole optimization (in a few slides).

What are these optimizations?

Copy propagation

```
int x = y;
... x ...
... x ...
```

Replace uses of `x` by `y` (and possibly remove `x`).

Common subexpression elimination

```
int x = a + b * c;
... a + b * c ...
```

Replace second occurrence of expression by `x`.

Constant folding

```
... 3 + 7 ...
... 5 * 8 ...
```

Compute constant expressions during compilation.

Inlining

```
int getX() { return x; }
... getX() ...
```

Replace call by `x`, avoiding call overhead.

Proper algorithms (and preconditions) discussed after Easter.

Challenges for Java JIT compilers

- Long-running loops: Need to change from interpreted to compiled code during execution.

  These have different stack layout, so must change stack frame when changing to compiled code.
- Deoptimization: Class loading may invalidate compiling assumptions; e.g. some method call cannot be determined statically. Need to go back to interpretation.

  Back to old stack representation; e.g. must add stack frames for inlined methods.

Java HotSpot server compiler

Basic features

Adds many more optimizations (discussed after Easter).

Another, SSA-based intermediate representation.

Phases: parsing, machine-independent optimization, instruction selection, code motion, register allocation, peephole optimization, code emission.

Further improvements

Start with interpretation.

When code deemed hot, perform client compilation.

When red hot, perform server compilation for cruising speed.

Garbage collection, general

The problem

- Lifetime of heap objects difficult to determine (pointers, aliasing).
- Not recycling unreachable objects (**memory leaks**) can lead to heap exhaustion.
- Recycling reachable objects leads to program errors.
- Recycled heap space often fragmented, leading to slower allocation.

Towards a solution

- Automate heap management: garbage collector reclaims unreachable objects.
- Necessary first step: identify reachable objects by following pointers from program roots.
- Many variations:
  – stop-the-world vs. concurrent.
  – age-neutral vs. generational.
  – copying vs. free-list based.

Garbage collection for Java, 1

HotSpot collection

- Heap divided into three **generations**:
  - Young generation. Newly created objects.
  - Old generation. Objects that have survived a number of collections are promoted here.
  - Permanent generation. Internal, heap-allocated data strucures (not collected).
- Allocation uses separate allocation buffer per thread. Fast/slow paths.
- Young generation. Three areas: Eden, where objects are created, and two alternating survivor spaces.
  Whenever Eden is filled, stop-and-copy collection from Eden and active survivor space to other survivor space.
- Old generation. Default is mark-and-sweep, stop-the-world collector.

Garbage collection for Java, 2

Current trends

Continued rapid progress.

Parallel collectors, for shorter pause times and more efficient use of multiple processors, are becoming available.

Major conclusion

Garbage collectors in modern JVM:s manage memory more efficiently than you can do it explicitly.

Some consequences for the Java programmer

- Don't use public instance variables; add set/get methods.
  There is no runtime overhead; these calls are inlined.
- Don't make classes final for performance.
  Client compiler will do this analysis for you.
- Don't try memory (de-)allocation yourself.
  Allocation is inlined and fast; your deallocation will not be better than GC.
- Don't use exception handling for control flow.
  Exception objects expensive; but no cost of exception handling when not used.

Preview of code optimization: A.f

Source code

```
public static int f (int x) {
   int r = 3;
   int s = r + 5;
   return s * x;
}
```

Resulting code

```
.method public static f(I)I
.limit locals 1
.limit stack 2
   iload_0
   iconst_3
   ishl
   ireturn
.end method
```

Observations

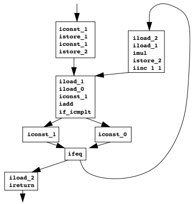
- r is initialized to 3 and never assigned to. Hence we can replace all uses by 3 and remove r.
- The expression 3 + 5 is computed by the compiler.
- s is also constant and can be replaced by 8.
- Multiplication by 8 is more efficiently done as left shift 3 positions.

We need algorithms to do this, not hand-waving!

A Jasmin example: fact

```
.method public static fact(I)I
.limit locals 3
.limit stack 3
    iconst_1
    istore_1
    iconst_1
    istore_2
  Label0:
    iload_1
    iload_0
    iconst_1
    iadd
    if_icmplt Label2
    iconst_0
    goto Label3
  Label2:
    iconst_1
  Label3:
    ifeq Label1
    iload_2
    iload_1
    imul
    istore_2
    iinc 1 1
    goto Label0
  Label1:
    iload_2
    ireturn
.end method
```


The control flow graph



Comments

- Code split into **basic blocks**.
- Control flow visualised by edges.
- Optimization simpler within a basic block.

Peephole optimization

A simple idea

Look at small sequences of instructions to find possibilities for improvement.

Can be iterated (**fixpoint iteration**), since one optimization may open new possibilities.

Use a suitable list type for your code (i.e. one that allows for fast deletion, insertion and reordering).

Easiest with pattern matching in Haskell.

Example (Constant folding)

```
bipush 7
bipush 5
iadd
```

can be replaced by just `bipush 12`.

More peephole optimization examples

Strength reduction

Replace an “expensive” operation (left) by a cheaper one (right)

`bipush 16` `imul`	`iconst_4` `ishl`
`ldc2_w 2.0` `dmul`	`dup2` `dadd`

Algebraic simplification

`iconst 0` `iadd`	
`iconst 0` `imul`	`pop` `iconst_0`

Further possibilities

Store/load elimination

Can the instruction pair

```
istore_0
iload_0
```

be removed?

Only if variable 0 not used later in method.

How big must the peephole be in order to give the best code in `f` to the right?

Code generated by `javac`

```
.method public static f(I)I
.limit locals 3
.limit stack 2
   iconst_3
   istore_1
   iload_1
   iconst_5
   iadd
   istore_2
   iload_2
   iload_0
   imul
   ireturn
.end method
```


Unreachable code

Java disallows “unreachable code”

Unreachable (or dead) code, i.e. code that can never be executed, is disallowed in Java.
However, to find all instances of dead code is an undecidable problem.
The languange specification defines a conservative approximation.

Peephole optimization can find some instances:

- Code after a goto and before next label,
- Code in a branch of an if or while statement with constant condition.

Also other jump-related optimizations, like jumping to the next instruction.

What next?

No more lectures on Submission A.

No lecture on Monday next week.

Lecture next Thursday starts with LLVM (target for Submission B).

After Easter: more LLVM, language extensions, code optimization.