Java Bytecode Peephole Optimisations

Apinan Hasthanasombat, Daniel Gavrilov

Simple Folding

In simple folding, we look for patterns of instructions that we can evaluate statically. We use regular expressions and BCEL's `InstructionFinder` to find these, and we evaluate them and store the result, if there is one.

Specifically, we look for **unary expressions** (constant instruction followed by an operator) and **binary expressions** (two constant instructions followed by an operator). By **constant instructions**, we mean instructions that push onto the stack:

- A constant stored in the memory allocated to the instruction
  `bipush sipush dconst fconst iconst lconst`
- A constant from the constant pool
  `ldc ldc2_w`

Unary operations

Unary operations we optimise are **negations** and **type conversions**. We optimise by:

1. Taking the value of the first instruction (the constant) and casting it to the type the second instruction (the operator) expects.
2. Applying the operation to the value and creating a PUSH instruction instance with the result.
3. Inserting the newly created instruction before the constant and deleting the constant and operator instructions.

We do not do any **type checking**, as we expect this to be done before this stage and we cast the constant to the expected type of the operator.

The **PUSH** class creates the most efficient push instruction given the value. For example, if passed a float value of `2.0`, it would create an `fconst_2` instruction, but `3.0` would create an `ldc` instruction and create a constant in the constant pool (or reuse one already there).

In case there are targeters to the first instruction (the constant), we redirect them to the inserted push instruction, but, we avoid optimising if there is a targeter to the second

instruction (the operator) as this would change the semantics of the program because we don't know what value on the stack the operator is acting on.

Binary operations

Binary operations we optimise are **arithmetic operations** (addition, subtraction, multiplication, division, modulo) **bitwise operations** (and, or, xor), **shifts** (left, right, logical right) and **comparison operations** (double, float and long comparisons). We optimise similarly to unary operations:

1. Taking the values of the first and second instructions (the constants) and casting it to the types the third instruction (the operator) expects.
2. Applying the operation to the two values and creating a PUSH instruction instance with the result.
3. Inserting the newly created instruction before the first constant and deleting the two constant instructions and the operator instruction.

As for unary, we don't do any type checking and cast the value of the constants to the type that the operator expects. We also don't optimise in case either the second instruction (the constant) or the third instruction (the operator) have targeters, as this would change the semantics of the program.

Conditional branches

We optimise unary and binary **integer comparison** instructions that conditionally redirect the flow of the program depending on the result of the comparison. We optimise them by:

1. Getting the values of the constant instructions (one in unary comparison, two in binary comparison) and performing the comparison
2. Depending on the result, we determine whether we need to follow the target of the comparison instruction
3. If we need to follow the target, we insert a `goto` instruction before the first instruction (the constant) with the same target as the comparison instruction, otherwise we do nothing
4. Finally, we delete the constant instructions and the comparison instruction

From this, we can see that in integer comparison of two constants, we either delete them all or replace them with a single `goto`.

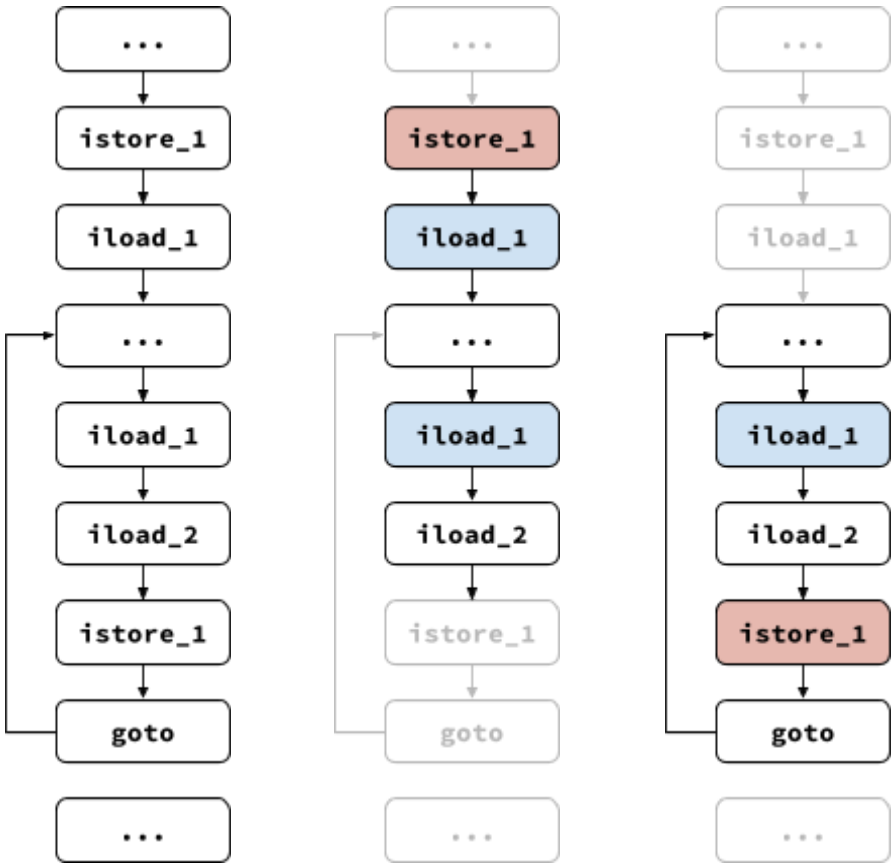
As previously, we do not perform the optimisation if the second constant instruction or the comparison instruction have a targeter, and we redirect any targets from the first instruction to the newly added `goto` instruction.

Variable Propagation

Determining reach

We start by determining what `load` instructions every `store` instruction reaches—or—for which `load` instructions the local variable assignment of a `store` instruction could be valid for.

For example, in the following control flow graph of a program:



The assignment of the first `istore_1` can reach the first and second `iload_1`, and the assignment of the second `istore_1` can reach only the second `iload_1`. Note that `iload_2` isn't reached by either of the store instructions, as it has a different index, referencing a different local variable.

We store these dependencies in two instances of the same data structure with type `Map<InstructionHandle, Collection<InstructionHandle>>` (essentially, a one-to-many store of `InstructionHandle`). One of them — we'll call it `storeReachingMap` — holds `store` instructions as its key, keeping track of the `load` instructions each store reaches, and the other — `loadReachingMap` — holds `load`

instructions as its key, keeping track of the `store` instructions each load is reached by. One can be derived from the other, but we use two for performance purposes. For example, for the program in the control flow graph above, they look like this:

```
istore_1: {iload_1, iload_1}
istore_1: {iload_1}
```

storeReachingMap

```
iload_1: {istore_1}
iload_1: {istore_1, istore_1}
```

loadReachingMap

We treat `iinc` as both a `store` and `load` instruction, as it loads the local variable, increments it and then stores it at the same index. As such, it can "reach" itself.

We populate the maps by:

1. Creating a control flow graph of the program using BCEL's `ControlFlowGraph`.
2. For each `store` instruction we find in the instruction list, we:
   a. Add it as a key in `storeReachingMap`
   b. Traverse the control flow graph (depth-first), starting at the `store` instruction and terminating when reaching another `store` instruction with the same index. For every `load` instruction we visit during the search, that has the same index, we:
      i. Append it to the collection under the `store` instruction key of `storeReachingMap`.
      ii. Append the `store` instruction to the collection under the `load` instruction key of `loadReachingMap`.

Replacing instructions

Once we have the two maps — `storeInstructionMap` and `loadInstructionMap` — we iterate through all the `store` instructions again. For each, we:

1. Do not optimise if it is an `iinc` instruction *or* it has a targeter to it *or* is not preceded by a constant.
2. If all `load` instructions it reaches are reached by this and only this `store` instruction, we:
   a. Replace all `load` instructions with a copy of the constant instruction that precedes the `store` instruction.
   b. Delete the `store` instruction and constant instruction.
3. If the `store` instruction reaches no `load` instructions (a **dead store**), the `store` and constant instruction preceding it are simply deleted.

An example with changes highlighted in **bold**:

```
iconst_1
istore_1
iload_1
iconst_2
iadd
iload_1
idiv
```

unoptimised bytecode

```
iconst_1
iconst_2
iadd
iconst_1
idiv
```

optimised bytecode

In the exceptional case where the `load` instruction is an `iinc` instruction, the `iinc` instruction is replaced by a constant followed by a `store` instruction of the same index. The value of the constant is incremented by the increment of the `iinc` instruction.

In the following example, `iinc` increments index 1 by 4, and since index 1 stores the value 1, the new value stored by iinc would be 5:

```
iconst_1
istore_1
iconst_1
iconst_2
iadd
iinc 1,4
```

unoptimised bytecode

```
iconst_1
iconst_2
iadd
iconst_5
istore_1
```

optimised bytecode

Eliminating dead code

With the integer comparison optimisation, we can end up with **dead code** and redundant `goto` instructions that simply target the next instruction. In the final pass, we get rid of these by:

1. Constructing a control flow graph (using BCEL's `ControlFlowGraph`).
2. For each instruction in the program, we:
   a. Delete it if it's not a node of the control flow graph.
   b. Delete it if it's a `goto` to the next instruction.
   c. Redirect any targeters to the next instruction.

Looping

The above mentioned optimisations are performed in a loop, so the code gets iteratively optimised until no further optimisation is possible. A simple example of why this is desirable is shown below:

```
public int methodOne() {
    int a = 62;
    int b = (a + 764) * 3;
    return b + 1234 - a;
}
```

First the variable a is propagated through the method:

```
public int methodOne() {
    int a = 62;
    int b = (62 + 764) * 3;
    return b + 1234 - 62;
}
```

Variable b is then folded (simple folding):

```
public int methodOne() {
    int b = 2478;
    return b + 1234 - 62;
}
```

Variable b is then propagated (in another iteration of the loop) and eliminated (unused variable):

```
public int methodOne() {
    int b = 2478;
    return 2478 + 1234 - 62;
}
```

And with another simple fold we have:

```
public int methodOne() {
    return 3650;
}
```