COMP207P - Compilers

Group 41 - Code Optimisation Report

Jack Rowland, Diana Lee

Constant Folding Peep-hole Optimisation

We first used the in-built Java bytecode disassembler to inspect the bytecode to see how best to optimise for each subgoal. Where possible, we try to cut out all store, unnecessary ‘load, and unnecessary push instructions. If a value being stored to register can be evaluated as `int, long, float` or `double` (i.e. a number, the required type), we try to manipulate it using byte-related instructions (`bipush, sipush`), or generate it into and consequently load it as a constant `#index` from the constant pool (using `ConstantPoolGen` and `ldc` variants). Using the provided `for` loop around the `optimizeMethod` method, we iterate through each `InstructionHandle` value of `InstructionList`, and optimise for the 3 sub-goals. `optimizeMethod` fetches the bytecode instructions for the requested method, then takes the bytecode data to initialise `InstructionList`. By declaring a new method based on a given method, we can obtain and manipulate the constant pool to optimise via constant folding.

Subgoal 1: Simple Folding

To achieve the first subgoal, the algorithm iterates through the instructions until it finds a stack operation, which it then runs on the last two items pushed onto the stack. Subsequently, it deletes all three of these instructions (the stack operation and two pushes), generates the result as a constant `#index` on the constant pool, and is thus replaced with a single instruction, to be loaded (`ldc`) from constant pool on to the stack. Arithmetic Instructions are handled first.

```
if (lastValue != null) {
        if (lastValue instanceof Integer) {
                i = CPGen.addInteger((int) lastValue);
                 instList.insert(handle, new LDC(i));
        }
        // … repeated for float, double and long
}
```

Operators that add nothing to the stack (e.g. NOP) are removed or replaced where needed. This builds the foundation for the rest of the algorithm, performing constant folding on the numbers (`int, long, float, double`) in the (given/artificial) bytecode constant pool.

Subgoal 2: Constant Variables

We build upon the initial algorithm to optimise for constant variables; local variables of the required numerical type whose values remain unchanged throughout method scope and are never reassigned.

Constant folding is achieved for constant variables by cycling through the instructions in the bytecode until a `StoreInstruction` is found. If the value being stored is calculable (i.e. can actually be stored in memory) then the constant folding algorithm takes effect. It loops through the remaining instructions, until it finds either the end of the instruction list, or another `StoreInstruction` to the same register.

Whilst looping, it replaces all load instructions for that register, for all numerical types (`int, double, float, long`) by a `LDC` or `LDC2_W` instruction on the value being loaded. Upon exit of the loop, the `StoreInstruction` is deleted using the method `private void instDel(InstructionList instList, InstructionHandle handle)`. Finally, the simple folding algorithm is applied to further optimise it.

```
if (handle.getInstruction() instanceof StoreInstruction)
{
        Number lastValue = getLastPush(instList, handle);
        if (canStore(instList, handle, lastValue))
        {
                handleStore(instList, handle, lastValue);
                InstructionHandle toDelete = handle;
                handle = handle.getNext();
                instDel(instList, toDelete);
                instList.setPositions();
        }
        else
        {
                handle = handle.getNext();
        }
}
```

Subgoal 3: Dynamic Variables

For dynamic variables, local variables which are re-assigned with a different constant number, we can build upon the instructions used to optimise for constant variables; An obvious example is `methodOne`, which adds a re-assignment into the method. We will assume each method contains variables of the same numerical type (in given methods, only `int` is given). In the case of the loop in `methodFour`, we focus on optimizing the variables `(b-a)`and ignore unrolling and folding of the loop.

Any instructions containing "`const`", "`load`", or "`store`" are replaced with "`ldc`" or "`bipush`"/"`sipush`" which loads a constant index or pushes a byte/short onto the stack. We want the values to be part of the constant pool, where possible, for minimum compile time, so we include handlers for any logical instructions (substitutions/assignments, xor, etc.), to be handled at relevant intervals. Handlers for the other three numerical types (float, long, double) are included as well.

Related notes

Numerical Type Conversions are handled upon calculating the last value in the stack in private number getLastPush(instList, handle), using a handler for ConversionInstruction classes and the method convert(lastOp,num).

For the Dynamic Variable tests, a stackmap frame error emerges; `if_icmpge` within the `println` statements and the loop raises an exception which expects a stackmap frame in a specific branch target (in our case, it's 14 or 37). As noted on Moodle, this is an issue raised by version 50 (`cgen.setMajor(50)`). We initially tried to deal with this by creating a handler for `IfInstruction` classes, but quickly realised that was out of scope for the assignment.