COMP0012 CW2 Report - Team 06

Ninaad Kulkarni, Darius Chitoroaga, Mert Albeyoglu

April 16, 2025

0.1 Simple Folding Optimisation

Objective:
The goal of the Simple Folding optimisation is to simplify bytecode by detecting and replacing constant arithmetic operations with their precomputed result. This reduces the number of instructions executed at runtime and demonstrates one of the core principles of peephole optimisation: constant folding. Our optimiser targets basic arithmetic operations (`+`, `-`, `*`, `/`) for primitive types: `int`, `long`, `float`, and `double`.

Overview of the Approach:
The optimiser works by scanning the bytecode of each method in a class file to find sequences of instructions where constant values are loaded, followed by an arithmetic operation. For instance, the sequence:

- `ldc 67`
- `ldc 12345`
- `iadd`

can be optimised to:

- `ldc 12412`

This reduces the instruction count and simplifies the stack manipulation required during execution.

Implementation Details:
The optimisation is implemented in the class `comp0012.main.SimpleFoldingFolder`. The main logic resides in the `optimize()` method, which is responsible for processing each method in a class, analysing its instructions, and rewriting optimisable patterns.

- **Step 1 – Bytecode Parsing:**
  We begin by using Apache BCEL to load the `.class` file with a `ClassParser`, generating a `ClassGen` and `ConstantPoolGen` to allow modification of the bytecode and the constant pool.

- **Step 2 – Instruction Pattern Matching:**
  For each method, we extract the `InstructionList` and use BCEL's `InstructionFinder` to identify patterns matching constant loads followed by arithmetic operations. Specifically, we look for the following instruction sequences:

  - `LDC LDC IADD`, `ISUB`, `IMUL`, `IDIV`
  - `LDC2_W LDC2_W LADD`, `LSUB`, `LMUL`, `LDIV`
  - `LDC LDC FADD`, `FSUB`, `FMUL`, `FDIV`
  - `LDC2_W LDC2_W DADD`, `DSUB`, `DMUL`, `DDIV`

  These patterns correspond to arithmetic operations on `int`, `long`, `float`, and `double`.

- **Step 3 – Constant Folding Logic:**
  Once a pattern is found, we extract the constant values involved using helper methods `getNumber()` and `computeFoldedResult()`. These methods determine the numeric type of the constants and safely compute the result of the operation, ensuring type preservation and checking for potential division by zero.

- **Step 4 – Instruction Replacement:**
  We replace the original three-instruction sequence with a single `LDC` or `LDC2_W` instruction representing the computed result. Any removed instructions are safely deleted from the instruction list, and BCEL's `TargetLostException` is handled to redirect any jump targets.

- **Step 5 – Method Update:**
  If any folding occurred, we update the method's max stack and local variable usage, then replace the old method in the `ClassGen` with the optimised version.

Summary:
The `optimize()` method in `SimpleFoldingFolder.java` encapsulates the complete logic for Simple Folding. This transformation maintains semantic correctness while reducing bytecode size and complexity. It is tested against handwritten Jasmin bytecode to ensure that the optimiser folds expressions that would normally be precomputed by the Java compiler, validating its correctness and effectiveness. This method was developed and tested on macOS Sonoma 14.2.1.

0.2 Constant Variables Optimisation

Objective:
The goal of the Constant Variables optimisation is to perform constant folding for the values of type `int`, `long`, `float`, and `double`, whose value does not change throughout the scope of the method. After the declaration, the variable

is not reassigned. The initial value must then be propagated throughout the method. To achieve this we must identify all constant variables then propagate the constant value.

Overview of the Approach:
The optimiser does a single pass through each method's bytecode, keeping track of a Stack of constants and a HashMap of variables.

- It takes the instruction list for the method and the method generator.
- Iterates through each instruction handle in the list, ignoring empty instructions.
- If an instruction is an instance of `IINC` (an increment instruction) or similar, we mark it as modified so we can ignore it.
- Previously, we defined a list of instruction handlers on a case-by-case basis that optimise specific instructions.
- We iterate through each handler, checking if it can be used in this case. If we can use it, then break from the loop.

Implementation Details:
The handlers we use are each placed in a HashMap as the value, with the corresponding instruction type as the key. This means that we can easily find the correct handler for each instruction type in `O(1)` time (i.e. `IADD` and `DADD` each would point to the `AddOperation` handler).

The implementation of the handler is simple in that it checks the type of the input `int`, `long`, `float`, or `double`, then extracts the value and operates. This output is then used to replace the arithmetic operation in the bytecode with a load instruction.

Additionally, there are handlers for Load, Store and Goto instructions.

The Load handler works by replacing the load instruction with a `LDC` or `LDC2_W`, depending on the type of the value (i.e doubles and longs into `LDC2_W` and ints and floats into `LDC`), then deleting the Load instruction.

The Store handler with make sure that what is being stored is not null, stores the instruction in the locally defined variables' hash map with the instruction index as the key, then deletes the instruction. This makes sure that we can propagate any of our constants later if we can. If not, then we replace the store as it was (i.e. if it was modified).

The Goto handler makes sure we don't have any redundant instructions; if the goto is to the next instruction, it gets deleted.

Results and Analysis:
This approach works well and does remove some instructions for constant variables (i.e. `a = 14 + 12; b = a+3;` to `a = 16; b = 19;`) from the bytecode, making it shorter, but the actual benefits do not manifest in any reasonable fashion; there is very little, if any change in the runtime when running the tests, so it is hard to tell if a real impact is made. The implementation is rather easy

to follow, each instruction has a handler which optimises specific instructions. However, some handlers have different side effects from others, which can make it difficult to reason about. Having a consistent interface and set of side effects for the handlers would be useful.

The code was tested on NixOS x86_64 Linux with openjdk_23.

0.3 Dynamic Variables Optimisation

Objective:
The goal of the Dynamic Variables optimisation is to identify and optimise the use of local variables of type `int`, `long`, `float`, and `double` whose values are reassigned with constant numbers within a method. Rather than treating these variables as non-constant throughout, our optimiser tracks their constant values from an assignment until the next update, and replaces subsequent load instructions with the corresponding constant loads. This enables further arithmetic folding and reduces redundant runtime computations.

Overview of the Approach:
Our optimiser performs a single sequential pass over each method's bytecode. Two key data structures are maintained:

- A **constant stack** that records numeric constants pushed by instructions.
- A **variable mapping** (named `dynVars`) that associates local variable indices with their latest constant value.

When a store instruction such as `ISTORE` is encountered, the top value from the constant stack is recorded in `dynVars`. Later, when a load instruction such as `ILOAD` for that variable index is detected, it is replaced with a constant-loading instruction (`LDC` or `LDC2_W`). Additionally, arithmetic operations that can be computed using two constants on the stack are folded into a single constant. In cases where control flow might disrupt the propagation, the constant stack is cleared to maintain correctness.

Implementation Details:
The dynamic variables optimisation is integrated into the overall constant folding framework within `comp0012.main.ConstantFolder`.

The key steps include:

- **Maintaining Propagation Data:**
  As the bytecode of each method is traversed, instructions pushing constants are pushed onto a temporary stack. When a numeric store instruction is encountered, the constant is popped and recorded in `dynVars`. This is a mapping from variable index to its current constant value.
- **Replacing Loads with Constants:**
  If a load instruction is found and the variable's index has an associated constant value in `dynVars`, the load is replaced with a corresponding constant-

loading instruction. This ensures that operations using this variable can be computed at compile time.

- **Arithmetic Folding:**
  When an arithmetic instruction (e.g., `IADD`, `ISUB`) is encountered and there are at least two constant operands available from the stack, the optimiser computes the result and replaces the entire arithmetic sequence with a single `LDC` or `LDC2_W` instruction.
- **Handling Control Flow:**
  To preserve correctness when the control flow may change, the constant stack is cleared, ensuring that only contiguous, stable intervals of constant values are propagated.

Results and Analysis:
Our unit tests (specifically, those in `DynamicVariableFoldingTest`) passed successfully, confirming that the dynamic variable optimisation is applied where possible. While the disassembled bytecode of `DynamicVariableFolding` in our test suite appears largely unchanged, the optimiser is capable of performing further folding in cases with more complex control flow or less aggressive compiler pre-folding. We predict that this is mostly because the Java compiler already folds many simple constant expressions. In scenarios where a variable is reassigned with a constant and then used later, our approach replaces redundant load instructions with direct constant loads. Overall, this reduces the number of arithmetic operations executed at runtime.

Summary:
The Dynamic Variables optimisation enhances the overall constant folding process by tracking and propagating variable values over intervals in which they are unchanged. This not only reduces the runtime instruction count by replacing variable loads with constant loads but also enables further arithmetic folding. Our approach has been verified through unit testing and detailed bytecode inspection. This ensures both semantic correctness and performance improvements. This method was developed and tested on macOS Sequoia 15.3.2 (24D81).

Conclusion

In this coursework, we implemented three bytecode optimisation strategies: Simple Folding, Constant Variables Optimisation, and Dynamic Variables Optimisation. Each approach was developed in its own module (`SimpleFoldingFolder.java` for Simple Folding, `ConstantVarFolder.java` for Constant Variables Optimisation, and `DynamicVarFolder.java` for Dynamic Variables Optimisation). Our final submission focuses on the Dynamic Variables Optimisation code, which also invokes the logic in the simple constant folder.

Although the integration of the constant variables optimisation encountered platform-specific constraints on Linux, this challenge provided us with the opportunity to build on each other's work. Our collaborative development process allowed us to refine our implementations, merge ideas, and enhance overall functionality, while each team member contributed complementary expertise to the project.

Overall, our combined approach not only demonstrates the potential of compile-time optimisations using Apache BCEL but also highlights the strength of collaborative development in addressing and overcoming technical challenges.

Platform and Contributions

Tested Platforms:

- macOS Sonoma 14.2.1
- macOS Sequoia 15.3.2 (24D81)
- NixOS x86_64 Linux

Team Contributions:

Ninaad Kulkarni implemented *Simple Folding*, which detects constant arithmetic operations in bytecode and replaces them with precomputed results. This reduces instruction count by folding operations like `ldc 5`, `ldc 3`, `iadd` into `ldc 8`.

Darius Chitoroaga worked on *Constant Variables*, identifying variables that are assigned only once. Their values are propagated throughout the method, enabling further constant folding wherever these variables are used, improving code efficiency and reducing redundancy.

Mert Albeyoglu developed the *Dynamic Variables* optimisation, which tracks variables that are reassigned with different constants. The optimiser replaces usages with their known constant values between assignments, applying folding in variable-stable intervals.

Dexin Fu, our fourth team member, had a EC, and was therefore unavailable to assist in this coursework.