Algorithm

Task 1: Simple Folding

The algorithm for constant optimisation is mainly implemented in ConstantFolder.java, under ConstantFolder class. The public method ConstantFolder is kept to generate a parser and class object and try the parse process. A private helper method getConstantValue() is introduced to distinguish between LDC and LDC2_W instructions. These two bytecode instructions represent the loading of constant values from the constant pool onto the operand stack. LDC is used for int and float, while LDC2_W is for long and double. This method extracts the numeric values accordingly.

The main method to compute arithmetic for constants is computeArithmetic, which is set as private as well. The method uses a switch sentence to separate different cases for integer, float, long and double. Each set of cases applies the same basic logic. The operator codes, such as IADD, ISUB, IMUL and IDIV for integers, are short values; the operator would be a parameter opcode to be passed and paired with them. Under each flag, the method will return related calculations. For example, in the IADD case, the method will return v1.intValue() + v2.intValue(). A special handling for divisions would be that if v2 is 0 then return null since 0 as divisor would be mathematically meaningless. For bytecode instruction generation, a private method createFoldingInstruction is implemented. It checks the data type of the obtained result and returns a related instruction, LDC type for integer and float, and LDC2_W for long and double, respectively.

The publicly optimised method is modified to run the optimisation for all four types of constants. To begin with, a ClassGen is created for the original class that is being optimised. The related ConstantPoolGen is created as well. To break the class down, we check it method by method using MethodGen. For each individual method, there would be a related InstructionList and InstructionFinder. Also, a string pattern is created including the data type and opcode that we are looking for. By running a nested for loop using iterator i, we can find the constant values and related opcode from the given instruction. Then we compute the arithmetic calculation to get the result, add a new instruction for that and try to delete the old one. The last thing to do before wrapping things up would be to reset the max value of the stack to correct the stack size. Last but not least, a public method write is kept in the class to process optimisation with related file paths.

Task 2: Constant Variables Optimisation

The constant variables optimisation identifies local variables whose values remain constant throughout a method's execution and replaces all uses of these variables with their literal values. This optimisation extends task 1's constant folding by enabling additional folding opportunities after variable values are propagated.

The algorithm for Task 2 operates in three main phases. First, the algorithm identifies local variables that are assigned exactly once in the method. This is crucial because variables that

are reassigned cannot be considered constants. Second, for variables with single assignments, the algorithm determines their constant values by examining the instruction that pushed the value onto the stack before it was stored. Finally, the algorithm replaces all loads of identified constant variables with their literal values throughout the method body.

The constant variable optimisation is implemented in the ConstantFolder class through two primary methods coupled with integration into the main optimisation cycle.

The implementation uses a two-pass scanning approach to identify constant variables. The first pass traverses the bytecode instructions to track variable assignments. It builds a map of variable indices to boolean flags indicating whether a variable has been modified more than once. The second pass identifies the actual constant values by examining instructions that precede store operations. The implementation handles various constant types (int, long, float, double) by detecting different bytecode instructions (LDC, LDC2_W, BIPUSH, SIPUSH, *CONST). This approach ensures that only variables with exactly one assignment are considered for optimization, maintaining program correctness.

After identifying constant variables, the implementation replaces all occurrences of these variables with their literal values. The method scans for load instructions (ILOAD, LLOAD, FLOAD, DLOAD) that reference constant variables. When found, it creates an appropriate constant-loading instruction based on the variable's type. It carefully handles instruction replacement, ensuring any branch targets pointing to the original instruction are updated to maintain control flow integrity.

An important safety measure is implemented to avoid applying this optimisation to methods with complex control flow: the implementation checks for branch instructions and skips optimisation for methods containing them, preventing potential issues with variable lifecycle.

Task 3: Dynamic Variables

The dynamic-variable folding logic is implemented in the ConstantFolder class within the private method optimizeDynamicVariables. This method is called from the primary optimise loop—alongside the simple-folding and constant-variable-folding routines—and is specifically responsible for propagating literal values of local variables across their uses, even when those variables are reassigned, while ensuring that each constant is applied only during the interval between its assignment and the subsequent update.

First, the optimizeDynamicVariables method initiates a comparison-safety pass by scanning the entire InstructionList for IfInstruction instances whose opcodes correspond to integer comparison operations, specifically IF_ICMPEQ through IF_ICMPLE. After identifying a branch instruction, the two instructions immediately before it are examined. If either of these is a LoadInstruction (for example, ILOAD or LLOAD), its local-variable index is added to the comparisonVariables set. By marking these indices in advance, the method ensures that any subsequent loads of those variables are exempt from constant propagation, thereby preserving the original runtime semantics of conditional expressions like x < y or x >= y.

Subsequently, the optimizeDynamicVariables method performs an assignment-tracking pass by iterating through the instruction list a second time. It detects any occurrence of a ConstantPushInstruction (for example, BIPUSH, SIPUSH, ICONST, or LDC) immediately followed by a StoreInstruction (such as ISTORE or LSTORE). Upon detecting such a pair, the literal value is extracted from the constant-push instruction via the appropriate BCEL API and recorded in the hash-table, currentConstants, with the store instruction's local-variable index serving as the map key and the extracted literal as its associated value. Consequently, this map maintains, at every step of the traversal, the most recent constant value assigned to each local variable.

The core of the dynamic-variable folding algorithm is the propagation pass, which executes within the same for-loop in optimizeDynamicVariables. During this pass, each LoadInstruction is subjected to two checks: (1) its local-variable index must not be listed in the comparisonVariables set—thereby preserving original comparison semantics—and (2) the currentConstants map must contain a literal value for that index. If both conditions are satisfied, the optimiser generates a new constant-loading instruction, for 32-bit values (int or float), it creates an LDC via cpgen.addInteger() or cpgen.addFloat(), and for 64-bit values (long or double), it creates an LDC2_W via cpgen.addLong() or cpgen.addDouble(). This new instruction is inserted immediately before the original LoadInstruction, which is then removed. All InstructionTargeters are updated to reattach any control-flow branches or exception-handler references to the new instruction. Each successful replacement sets the modified flag to true, signaling the outer optimize loop to rewrite the method and, if necessary, perform further optimisation passes.

Lastly, optimizeDynamicVariables handles constant invalidation. Whenever it encounters a StoreInstruction that is not immediately preceded by a ConstantPushInstruction, it treats this as a runtime-computed assignment and removes the corresponding entry from the currentConstants map. By doing so, any subsequent LoadInstruction for that variable will not be replaced with a stale constant, accurately reflecting that the variable's value is no longer known at compile time.

By combining these four passes—comparison detection, assignment tracking, safe propagation, and invalidation—within optimizeDynamicVariables, the ConstantFolder.optimize() method fully realises the dynamic-variable folding sub-goal. Local variables may be reassigned multiple times, yet their literal values are still propagated where valid, without altering control-flow semantics.

Platform/OS Tested On

All development and testing have been performed on Ubuntu 22.04 LTS, using OpenJDK 17 and Apache Ant 1.10.x. In that environment, invoking ant in the project root compiles the source, executes all unit tests, and produces the optimised bytecode without errors. The only messages emitted are benign deprecation warnings from Ant's use of the Java security manager, but they do not interfere with compilation or test success.

Because Ubuntu 22.04 standardizes on Unix-style line endings (\n), the output of System.out.println(...) calls also uses \n, ensuring that tests which assert exact newline

characters pass reproducibly. In short, on Ubuntu 22.04 the entire build, optimisation routines (simple folding, constant-variable folding, dynamic-variable folding), and test suite execute cleanly and reliably.

Contributions

Berken was responsible for designing and implementing the dynamic-variable folding logic in and authored the corresponding report section describing the algorithm's design and integration. He also performed thorough testing of the complete optimiser on Ubuntu 22.04 LTS, and wrote the report subsection detailing the project's test platform and environment, ensuring reproducibility and clarity for the reader.

Zhaoqi implemented the simple folding routine, leveraging BCEL's InstructionFinder to recognise and collapse constant arithmetic sequences, and authored the corresponding report section that details the pattern-matching approach, result computation, and integration into the overall optimisation workflow.

Moiz developed the constant-variable folding functionality and authored the accompanying report section that describes how single-assignment locals are detected and propagated, including examples and test results.

Joshua did not contribute to this coursework; the work was divided equally among Berken, Zhaoqi, and Moiz.