COMP0012- COMPILERS
COURSEWORK PART II: CODE OPTIMISATION

Group 03
Academic Year 2023 - 2024

1. IMPLEMENTATION

Overview
The ConstantFolder class in the comp0012.main package leverages bytecode manipulation techniques to perform constant folding optimization on Java bytecode files. The primary goal is to simplify constant expressions and reduce computational overhead at runtime by preprocessing these expressions at compile time. This optimization process is structured to parse the original class file, identify optimization opportunities at the method level, and implement transformations instruction by instruction.

Parsing and Initial Setup
The optimization process begins with the instantiation of the ConstantFolder class, which initializes several key components:

1. ClassParser and ClassGen: These components from the Apache BCEL library are utilized to parse the existing .class file and generate a manipulable structure.

2. ConstantPoolGen: Manages the constant pool, a critical component in Java bytecode that stores literals and symbolic references.

3. Stack and HashMap Collections: Used to track the values loaded onto the stack and variables stored, facilitating the identification and manipulation of constant values and expressions.

Instruction Handling and Optimization Techniques
The optimization is conducted through a series of specialized handler methods, each responsible for a specific type of bytecode instruction:

1. handleLoad and handleStore: Manage the loading and storing of constant values. handleLoad pushes constants onto a stack when a constant-loading instruction (e.g., LDC, ICONST) is encountered. handleStore captures values from the stack into a hashmap when a store instruction is detected, tracking if they remain constant.

2. handleArithmetic: Identifies arithmetic operations involving constants. If both operands are constants (already on the stack), the operation is executed immediately using the do_calc method, which supports various data types and arithmetic operations. The original instructions are replaced with a new instruction that loads the computed result.

3. handleVariableLoad: Checks if a loaded variable is marked as constant in the hashmap and, if so, pushes its constant value onto the stack, bypassing the standard load process.

4. handleConversion and handleComparison: Apply direct conversions and comparisons to constants, replacing sequences of instructions with simplified ones that directly use the computed constants.

Method-Level Processing

Each method in the class file is processed individually through the method_process method. This involves:

1. Instruction Parsing: An InstructionList is generated for the method's bytecode.

2. Instruction Optimization: Each instruction is passed to the handle_instruction method, which delegates to specific handlers based on the instruction type, optimizing as needed.

3. Method Reassembly: After processing, a new method structure is assembled with optimized instructions, updating the stack and local variables' requirements.

Compilation and Output
After all methods are processed and optimized:

1. Class Reassembly: The class structure is updated with optimized methods.

2. Bytecode Generation: The modified class structure is compiled back into bytecode, and the optimized class file is written to the specified output path using write.

2. TEST ENVIRONMENT

The project is tested on
Windows 10 22H2,
Openjdk 16.0.2 2021-07-20
OpenJDK Runtime Environment Corretto-16.0.2.7.1 (build 16.0.2+7),
OpenJDK 64-Bit Server VM Corretto-16.0.2.7.1 (build 16.0.2+7, mixed mode, sharing), and
apache-ant-1.9.16 since the department Linux machines have no ant installed.

3. CONTRIBUTION

The contribution of each team member to the coursework is shown in the fig 1.

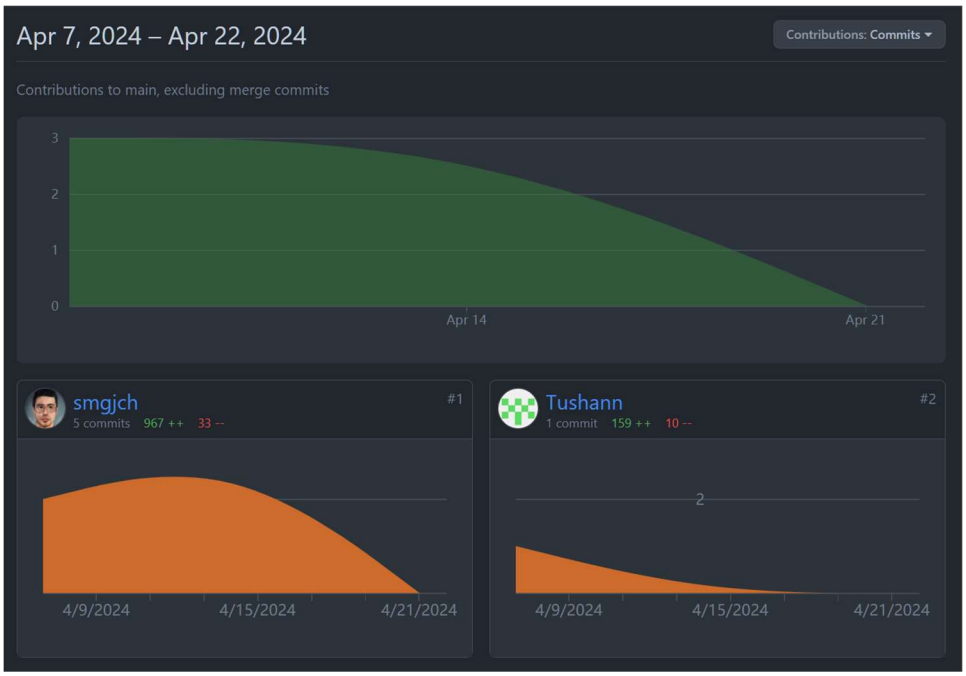

Fig 1. Contributions

Jucheng Hu 967++ 33—
Tushann Arya 159++ 10--