Pattern-Based Peephole Optimizations with Java JIT Tests

Zhiqiang Zang
The University of Texas at Austin
Austin, Texas, USA
zhiqiang.zang@utexas.edu

Aditya Thimmaiah
The University of Texas at Austin
Austin, Texas, USA
auditt@utexas.edu

Milos Gligoric
The University of Texas at Austin
Austin, Texas, USA
gligoric@utexas.edu

ABSTRACT

We present JOG, a framework that facilitates developing Java JIT peephole optimizations alongside JIT tests. JOG enables developers to write a pattern, in Java itself, that specifies desired code transformations by writing code before and after the optimization, as well as any necessary preconditions. Such patterns can be written in the same way that tests of the optimization are already written in OpenJDK. JOG translates each pattern into C/C++ code that can be integrated as a JIT optimization pass. JOG also generates Java tests for optimizations from patterns. Furthermore, JOG can automatically detect possible shadow relation between a pair of optimizations where the effect of the shadowed optimization is overridden by another. Our evaluation shows that JOG makes it easier to write readable JIT optimizations alongside tests without decreasing the effectiveness of JIT optimizations. We wrote 162 patterns, including 68 existing optimizations in OpenJDK, 92 new optimizations adapted from LLVM, and two new optimizations that we proposed. We opened eight pull requests (PRs) for OpenJDK, including six for new optimizations, one on removing shadowed optimizations, and one for newly generated JIT tests; seven PRs have already been integrated into the master branch of OpenJDK.

CCS CONCEPTS

• **Software and its engineering → Just-in-time compilers**; **Domain specific languages**; *Software testing and debugging*; *Formal software verification*; *Source code generation.*

KEYWORDS

Just-in-time compilers, code generation, peephole optimizations

ACM Reference Format:
Zhiqiang Zang, Aditya Thimmaiah, and Milos Gligoric. 2023. Pattern-Based Peephole Optimizations with Java JIT Tests. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598038

1 INTRODUCTION

Peephole optimization [21, 24] is an optimization technique performed on a small set of instructions (known as a *window*), e.g., `A + A` is transformed to `A << 1`. Popular compilers such as GCC, LLVM, and Java JIT, include dozens if not hundreds of peephole optimizations [3, 17, 29].

```
1  @Test
2  @IR(failOn = {IRNode.ADD})
3  @IR(counts = {IRNode.SUB, "1"})
4  // Checks (a - b) + (c - a) => (c - b)
5  public long test8(long a, long b, long c) {
6    return (a - b) + (c - a);
7  }
```

Figure 1: An example JIT test available in OpenJDK.

Traditionally, each peephole optimization is implemented as a *compiler pass*. Each compiler pass detects windows, i.e., a sequence of instructions that can be optimized, and replaces each window with an equivalent, albeit more efficient, sequence of instructions. These implementations are written in the language in which the compiler is implemented (C/C++ for Java JIT) and they leverage compiler infrastructure to detect instructions of interest. Representation of these instructions inside the compiler infrastructure is substantially different from code written in the programming language itself [19]. This disconnect introduces a burden on compiler developers to perform proper reasoning to detect windows of interest, to do the instruction mapping from high-level code (what developers write) to low-level code, and to document their intention. The process is tedious and error prone.

Alive [19] was an improvement over the traditional approach: a developer writes patterns in a domain specific language (DSL) over the *intermediate representation* (IR) of the program (LLVM bitcode) which are then translated into compiler passes. The DSL used in Alive is still very much disconnected from code written in the programming language being optimized (C++). This disconnect introduces a steep learning curve and lacks most of common software tools, e.g., an IDE. Alive also focused on C++ intricacies and undefined behavior.

Our insight is that *many peephole optimizations can be expressed in the programming language that is being optimized* (e.g., Java). We found the motivation in existing tests for Java JIT. Most tests for JIT optimizations in OpenJDK are written in Java and some of the tests contain Java code that follows specific *patterns* so as to trigger the optimizations under test [28]. Figure 1 shows such a JIT test from OpenJDK, which triggers the peephole optimization that transforms `(a - b) + (c - a)` into `c - b`, by returning `(a - b) + (c - a)` (line 6). Such a pattern expresses, in Java code, the window to be recognized by a specific optimization. *We propose to extend this concept to use the patterns not only to write tests but to express the entire optimization, including code before and after the optimization.*

We present JOG, the first framework that enables developers to write optimization patterns in a high-level language (Java). Namely, using JOG, a Java JIT compiler developer writes optimization patterns as Java statements. Patterns are type-checked (by the Java compiler) and automatically translated into compiler passes (by JOG). Additionally, Java tests for the optimizations can be automatically generated from the patterns. Writing patterns in Java for the Java JIT compiler ensures that sequences of statements are meaningful, i.e., windows can indeed appear in programs (which is not necessarily the case when matching intermediate representation or compiler abstractions). Next, writing patterns in Java simplifies the reasoning behind each peephole optimization: what used to be comments documenting an optimization inside the Java JIT compiler for dozens of lines of code, or what used to be a test that describes how to trigger an optimization, becomes a self-documenting pattern. Finally, while writing patterns in JOG, a developer can use software engineering tools available for the language (e.g., IDE, linter). Having patterns in Java also enables future application of program equivalence checkers that work on either Java code or bytecode (which can be easily obtained by compiling JOG patterns).

Furthermore, conciseness of patterns makes it easier to analyze relations between optimizations. JOG automatically detects possible shadow relation between a pair of optimizations where the effect of the shadowed optimization is overridden by another. Consider two optimizations X and Y: X transforms `(a - b) + (c - d)` into `(a + c) - (b + d)` and Y transforms `(a - b) + (b - c)` into `a - c`, where `a`, `b`, `c`, `d` are all free variables. Note that any expression matching `(a - b) + (b - c)` (X) also matches `(a - b) + (c - d)` (Y), which means X can be applied wherever Y can be applied, so the effect of X will shadow Y if X is always applied before Y in a compiler pass. JOG can automatically report the shadow relations.

Using JOG, we wrote a total of 162 optimization patterns, including 68 existing optimizations in OpenJDK, 92 new optimizations adapted from LLVM, and two new optimizations. Most of the patterns that we extracted from OpenJDK were existing tests of the optimizations, or they were hand-written as examples in the comments documenting the C/C++ implementation. Our most complex pattern has only 115 characters in contrast to the 462 characters of its C/C++ counterpart. Our evaluation shows that generating code from patterns using JOG does not reduce the effectiveness of JIT optimizations. We also identified a bug in existing Java JIT as one optimization was unreachable as a consequence of being shadowed by another.

Recently, we have opened a group of eight pull requests (PRs) for OpenJDK (six for new optimizations, one for fixing the aforementioned shadowed optimizations, and one for new JOG generated JIT tests of existing optimizations). Seven of the PRs were already accepted and integrated into the master branch; the remaining PR is under review. We intend on opening PRs on the remaining optimizations in the future.

The main contributions of this paper include the following:

- We present JOG, the first framework that allows developers to specify a Java JIT peephole optimization as a pattern written in Java itself, extending the existing approach to writing tests for JIT. The pattern is automatically translated into C/C++ code as a JIT optimization pass, and a Java test for the optimization is generated from the pattern as needed.
- JIT optimizations written in JOG is easier to read and understand. We translated 68 existing patterns in OpenJDK. The evaluation shows a 64% reduction in characters of code and a 53% reduction in the number of identifiers in code when writing optimizations in JOG relative to existing hand-written code in OpenJDK.
- Code generated from JOG maintains the effectiveness of JIT optimizations. The evaluation shows that the impact on performance is minimal on replacing existing hand-written code in OpenJDK with JOG generated code for existing patterns. We also wrote 92 new patterns adapted from LLVM. A total of six PRs on the new patterns were opened, of which five PRs have been integrated into the master branch of OpenJDK.
- We present an algorithm to determine if one optimization shadows another written in patterns using JOG. We ran the algorithm on all the translated patterns to detect shadows between patterns. We opened one PR on removing shadowed patterns that has been integrated into the master branch of OpenJDK.
- JIT tests generated from JOG complements existing test suites in OpenJDK. We generated tests for existing optimizations in OpenJDK and opened one PR to add 10 new tests for existing untested optimizations in OpenJDK.

We believe that JOG enables developers to quickly write and evaluate a large number of peephole optimizations by writing patterns in a familiar programming language and very much similar to the way the existing tests for JIT are written. JOG is publicly available at https://github.com/EngineeringSoftware/jog.

2 EXAMPLE

The IR test, written in Java using IR test framework [30], is a recommended approach in OpenJDK to testing JIT peephole optimizations. We already showed such a test in Figure 1. While the test runs, the method annotated by `@Test` (`test8`) is compiled by JIT, with the expression `(a - b) + (c - a)` optimized to `c - b`. Then the IR shape of the compiled method is checked against certain rules specified in `@IR` (line 2–3). The rules verify that the optimization from `(a - b) + (c - a)` to `c - b` indeed happens, by checking that the compiled method must not contain `ADD` node (line 2) and must have exactly one `SUB` node (line 3).

Using JOG, developers can write the optimization under test in the same way as in the already existing test. Figure 2a shows a pattern written using JOG that expresses the optimization, which is a Java method annotated with `@Pattern`. The parameters of the method declare all the variables (line 2 in Figure 2a), `a`, `b`, and `c`, that are used in the method body. Parameter type `long` indicates the data type involved in the optimization. Two API calls inside the method body, `before((a - b) + (c - a))` (line 3 in Figure 2a) and `after(c - b)` (line 4 in Figure 2a), specify the matched expression before the optimization and the transformed expression after the optimization, respectively. Both `before` and `after` API call are written in the same way as the existing test is written. `before((a - b) + (c - a))` directly uses existing test code from `return (a - b) + (c - a);` (line 6 in Figure 1), and `after(c - b)` is extracted from the comment `// Check (a - b) + (c - a) => (c - b)` (line 4 in Figure 1).

Because the pattern and the test are written in the same way, not only does JOG provide an intuitive way to express an optimization,

```
@Pattern
public void ADD8(long a, long b, long c) {
  before((a - b) + (c - a));
  after(c - b);
}
```

(a) Pattern written using JOG.

```
1   Node *AddLNode::Ideal(PhaseGVN *phase, bool can_reshape) {...
2     Node* in1 = in(1);
3     Node* in2 = in(2);
4     int op1 = in1->Opcode();
5     int op2 = in2->Opcode();
6     if (op1 == Op_SubL) {...
7       // Convert "(a-b)+(c-a)" into "(c-b)"
8  -    if (op2 == Op_SubL && in1->in(1) == in1->in(2)) {
9  +    if (op2 == Op_SubL && in1->in(1) == in2->in(2)) {
10        return new SubLNode(in2->in(1), in1->in(2));
11      }
12    }...
13  }
```

(b) Hand-written code (with bug) in OpenJDK.

```
Node *AddLNode::Ideal(PhaseGVN *phase, bool can_reshape) {...
  Node* _JOG_in1 = in(1);
  Node* _JOG_in11 = _JOG_in1 != NULL && 1 < _JOG_in1->req() ?
                    _JOG_in1->in(1) : NULL;
  Node* _JOG_in12 = _JOG_in1 != NULL && 2 < _JOG_in1->req() ?
                    _JOG_in1->in(2) : NULL;
  Node* _JOG_in2 = in(2);
  Node* _JOG_in21 = _JOG_in2 != NULL && 1 < _JOG_in2->req() ?
                    _JOG_in2->in(1) : NULL;
  Node* _JOG_in22 = _JOG_in2 != NULL && 2 < _JOG_in2->req() ?
                    _JOG_in2->in(2) : NULL;
  if (_JOG_in1->Opcode() == Op_SubL
      && _JOG_in2->Opcode() == Op_SubL
      && _JOG_in11 == _JOG_in22) {
    return new SubLNode(_JOG_in21, _JOG_in12);
  }...
}
```

(c) Code generated from JOG.

Figure 2: An example of a peephole optimization as implemented in OpenJDK and JOG, and associated test.

without writing any extra code, but also it can automatically generate the test from the pattern. First the `@Test` method declares exactly the same free variables as the pattern (`long a, long b, long c`), and returns exactly `before`’s expression in the pattern (`return (a - b) + (c - a);`). Next JOG analyzes `before((a - b) + (c - a))` and `after(c - b)` in the pattern, (1) to find in `after` the numbers of operators (one `SUB`) and (2) to find which operators disappear from `before` to `after` (`ADD`). JOG then maps the operators to the corresponding IR node types used in IR tests and makes `@IR` annotations (`@IR(counts = IRNode.SUB, “1”)` and `@IR(failOn = IRNode.ADD)`). Eventually the exactly same test as shown in Figure 1 can be generated from the JOG pattern.

More importantly, JOG automatically translates a pattern into the C/C++ code that can be directly included in a JIT optimization pass. Figure 2c shows the C/C++ code translated by JOG from the pattern, and Figure 2b shows the hand-written code extracted from OpenJDK, that implements the same JIT peephole optimization that

```
Pattern := MethodModifier* MethodHeader MethodBody
MethodHeader := "void" Identifier "(" FormalParameterList ")"
MethodBody := "{" Stmt* "}"
Stmt := BeforeStmt | AfterStmt | IfStmt | AssignStmt
BeforeStmt := before "(" expression ")" ;
AfterStmt := after "(" expression ")" ;
```

Figure 3: JOG Syntax. The non-terminals that are not defined in the figure share the same definition as Java [11].

transforms `(a - b) + (c - a)` into `c - b`. The implementation contains two steps: (A) match any expression that is of interest to the optimization and (B) return a new optimized equivalent expression. In this example, any matched expression satisfies the following four conditions: (1) the expression is an addition expression (implicitly line 1 in Figure 2b because the method works only inside an additive expression); (2) the left operand (`a - b`) is a subtraction expression (line 6 in Figure 2b); (3) the right operand (`c - a`) is also a subtraction expression (line 9 in Figure 2b); (4) the left operand of the left sub-expression (`a`) is equal to the right operand of the right sub-expression (`a` again) (line 9 in Figure 2b). After a match is found, the code constructs a new subtraction expression (`c - b`) using `b` and `c`, and returns it. The transformation reduces the cost of evaluating the expression by two operations, from two subtractions and one addition to only one subtraction. Interestingly, this code has a bug (in OpenJDK) because of wrong access to the right operand of the right sub-expression, which is supposed to be `in2->in(2)` while developers wrote it as `in1->in(2)`. It took 13 years to discover and fix the bug; line 8 was inserted in 2008 and had been not touched until 2021 [41]. If the optimization had rather been implemented using JOG, the bug could have been avoided.

JOG reads from `before` and `after` APIs the expressions to match and return, respectively. JOG analyzes the expressions to infer the conditions to check and to infer the new expression to construct, and eventually assembles everything in C/C++ code as the output. Figure 2c shows the code generated from the pattern in Figure 2a. The generated code keeps the same functionality while avoiding the bug in the hand-written code of Figure 2b.

3 JOG FRAMEWORK

This section describes the JOG framework in detail. We describe the syntax for writing patterns, semantics of the statements, translation details, and test generation from patterns.

3.1 Syntax

Figure 3 defines the syntax of JOG, which is a subset of Java (for non-terminals that are not defined in the figure, please refer to the Java grammar [11]). Every optimization is written as a method in Java, which we call a *pattern*. The method body contains several statements. Each statement can be `BeforeStmt`, `AfterStmt`, conditional or assignment. We introduce `BeforeStmt` to specify the expression that a pattern has to match, and we introduce `AfterStmt` to specify the optimized expression as a result of applying the optimization.

3.2 Semantics

The parameters of the method declare the variables used in the pattern. There are two types of variables: *constant* and *free*. A constant variable represents a literal (e.g., 42); a free variable represents any expression, including literals. A parameter declares a free variable unless explicitly declared as a constant variable.

The semantics of a pattern is a peephole optimization that transforms a certain set of instructions into another set of instructions. Thus, a pattern must have one `BeforeStmt` and one `AfterStmt`. Both statements contain an expression. The expression inside `BeforeStmt` defines the set of instructions that can be transformed by the optimization, and the expression inside `AfterStmt` defines the set of instructions as a result of the optimization. It is possible that the optimization is supposed to be applied only under certain preconditions. Any necessary precondition can be specified as the condition of an `IfStmt`, and either `BeforeStmt` or `AfterStmt` can be included in the "then" branch of such `IfStmt`. To ensure every pair of `BeforeStmt` and `AfterStmt` match with each other, `AfterStmt` must be either a sibling node of `BeforeStmt` after it (in sequential order) or a descendant of such a sibling node, e.g., `if ([COND]) {BeforeStmt} AfterStmt` is not a valid pattern because the `AfterStmt` is neither a sibling node of `BeforeStmt`, nor a descendant of such a sibling node.

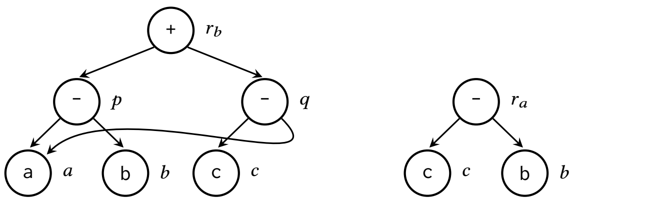
3.3 Translation

We implement JOG in the Java programming language and provide two API annotations, `@Pattern` and `@Constant`, and two API methods, `void before(int expression)` and `void after(int expression)` (`int` can also be `long`), to express a pattern. We also reuse Java constructs to make it easier to write a pattern, such as `if` statements and assignments.

A pattern is recognized by a method annotated with `@Pattern`. All variables used in the body of the method must be declared as parameters of the method. A parameter can be annotated with `@Constant` to indicate that the parameter represents a constant variable rather than a free variable. A valid pattern requires a `before` method call and an `after` method call in the method body and it may contain `if` statements for preconditions or assignments for local re-assignment of variables.

JOG starts translating a pattern by parsing the expression from `before` API and constructing an *eAST* (extended abstract syntax tree, which is strictly a directed acyclic graph) for the expression. During the construction, JOG maintains a map from identifiers in the expression, such as variables or number literals, to leaf nodes in the eAST. This map is then used to construct the eAST for the expression from `after` API or any preconditions, because JOG reuses the same node in `before` when seeing the same identifier in `after` API or preconditions, to ensure the correct transformation from `before` to `after`. Figure 4 shows the eASTs for the `before`'s and `after`'s expression of pattern ADD8 (Figure 2a).

JOG next translates eASTs into C/C++ code that can be included in a JIT optimization pass. As we have seen in Section 2, the generated C/C++ code consists of an `if` statement. The condition of the `if` statement is the conjunction of all the conditions that have to be satisfied for any expression to be matched by the pattern. The `then` branch of the `if` statement ends with a `return` statement that returns an optimized expression. JOG first traverses `before`'s eAST, and for every node in the eAST JOG translates the path from the root to the node into a pointer access chain in C/C++ (line 2–11 in Figure 2c). For example, node b in Figure 4a can be accessed



(a) eAST of `before` expression. (b) eAST of `after` expression.

Figure 4: eASTs for pattern ADD8 in Figure 2a.

by `in(1)->in(2)`. Note one node could be accessed in more than one way, and JOG always picks the smallest one in lexicographic order. Considering node a in in Figure 4a, which is both the left child of node p (`in(1)`) and the right child of node q (`in(2)`), this node can be accessed by both `in(1)->in(1)` and `in(2)->in(2)`, JOG translates the node into `in(1)->in(1)`. Next, JOG generates the conditions. JOG traverses `before`'s eAST again to generate operator check and possible constant check, for example, checking subtraction operators for node p, `_JOG_in1->Opcode() == Op_SubL` (line 12 in Figure 2c), and q, `_JOG_in2->Opcode() == Op_SubL` (line 13 in Figure 2c), where `_JOG_in1 = in(1)` and `_JOG_in2 = in(2)`. Also, JOG generates same-node check for any node that can be accessed in more than one way from the root. For instance, node a in Figure 4a results in the condition `_JOG_in11 == _JOG_in22` (line 14 in Figure 2c), where `_JOG_in11 = in(1)->in(1)` and `_JOG_in22 = in(2)->in(2)`. Additionally, if the pattern provided contains any `if` conditions, i.e., the specified optimization requires preconditions, JOG translates the eASTs of the preconditions into conditions in C/C++ code in the same way.

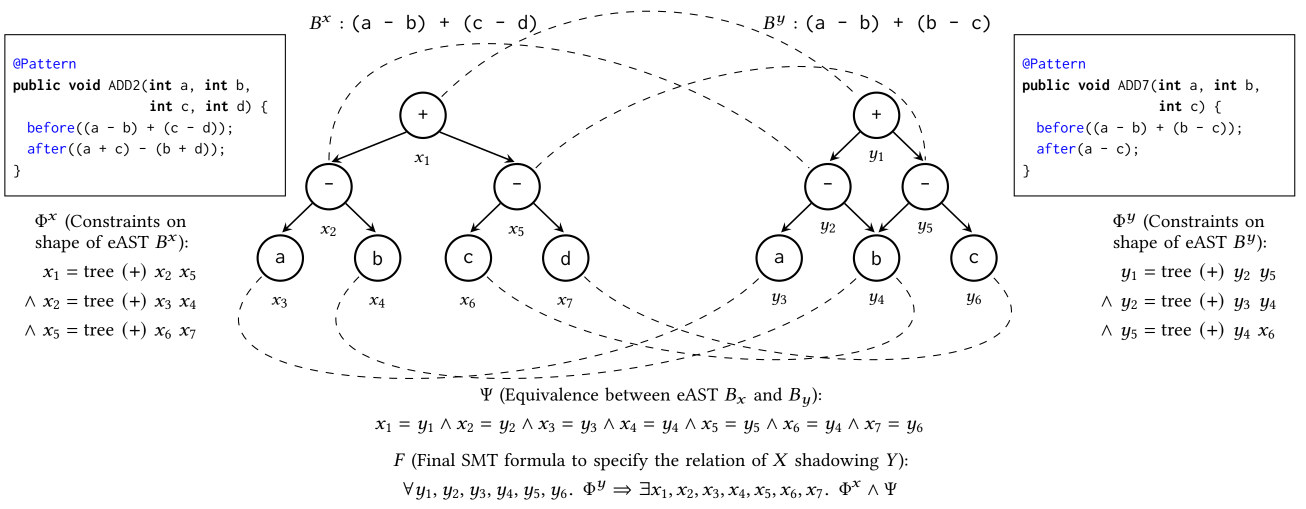
To translate `after`'s eAST, JOG performs a Depth-First Search (DFS). Every leaf node in `after`'s eAST is shared with `before`'s eAST so JOG reuses the pointer access chain for the node, i.e., `_JOG_in21` for node c and `_JOG_in22` for node b in Figure 4b. For an internal node in `after`'s eAST, JOG instantiates a new expression according to the operator of the node. For example, node r_a in Figure 4b leads to `new SubLNode(_JOG_in21, _JOG_in12)` (line 15 in Figure 2c). Finally JOG generates a return statement that returns the instantiation generated for the root node as translation of `after`'s eAST.

With `before`'s and preconditions' eASTs translated into conditions and `after`'s eAST translated into a return statement, JOG encapsulates them in an `if` statement (line 12–16 in Figure 2c) and prepend proper variable declarations (line 2–11 in Figure 2c). This concludes the translation of one pattern. When there are multiple patterns, JOG translate them in the order of the patterns written in the file provided.

3.4 Test Generation

Writing the pattern in the same way that the existing test is written allows JOG to generate an IR test from the pattern. We next describe the process of test generation using the example in Figure 1. Although the test is an already existing IR test in OpenJDK, JOG can generate exactly the same test from the pattern (Figure 2a).

The `@Test` method first declares exactly the same free variables as the pattern (`long a, long b, long c`), and returns exactly `before`'s



B^x : (a - b) + (c - d)
B^y : (a - b) + (b - c)
@Pattern
public void ADD2(int a, int b, int c, int d) {
  before((a - b) + (c - d));
  after((a + c) - (b + d));
}
@Pattern
public void ADD7(int a, int b, int c) {
  before((a - b) + (b - c));
  after(a - c);
}
Φ^x (Constraints on shape of eAST B^x):
x_1 = tree (+) x_2 x_5
$\wedge$ x_2 = tree (+) x_3 x_4
$\wedge$ x_5 = tree (+) x_6 x_7
Φ^y (Constraints on shape of eAST B^y):
y_1 = tree (+) y_2 y_5
$\wedge$ y_2 = tree (+) y_3 y_4
$\wedge$ y_5 = tree (+) y_4 y_6
Ψ (Equivalence between eAST B_x and B_y):
$x_1 = y_1 \wedge x_2 = y_2 \wedge x_3 = y_3 \wedge x_4 = y_4 \wedge x_5 = y_5 \wedge x_6 = y_4 \wedge x_7 = y_6$
F (Final SMT formula to specify the relation of X shadowing Y):
$\forall y_1, y_2, y_3, y_4, y_5, y_6.\ \Phi^y \Rightarrow \exists x_1, x_2, x_3, x_4, x_5, x_6, x_7.\ \Phi^x \wedge \Psi$

Figure 5: Illustration of shadow detecting algorithm on pattern ADD2 (X) shadowing ADD7 (Y). Every eAST node is represented by a variable, and the dashed lines connect equivalent nodes (e.g., x_1 is equivalent to y_1). The formula F shows the final SMT formula that specifies the shadow relation.

expression in the pattern (`return (a - b) + (c - a);`). One exception is that when the pattern has a constant variable (Section 3.3), JOG uses a random number to substitute the constant variable. Next JOG analyzes `before` and `after` in the pattern. JOG searches in `after`'s eAST (`c - b`) to count the number of operators (one `SUB`), and compares `before`'s and `after`'s eASTs to obtain the operators that exist in `before` but not in `after` (`ADD`). JOG then maps the operators to the corresponding IR node types used in IR tests and makes `@IR` annotations (`@IR(counts = IRNode.SUB, "1")` and `@IR(failOn = IRNode.ADD)`).

Our current implementation does not generate tests for the patterns with preconditions that specify invariants between variables. For example, a pattern rewritten from OpenJDK [39] that transforms `(x >>> C0) + C1` to `(x + (C1 << C0)) >>> C0)` requires a precondition `C0 < 5 && -5 < C1 && C1 < 0 && x >= -(y << C0)`. A random integer number would not be a good test input for constant variable `C0` or `C1` in the pattern because it cannot satisfy the precondition so as to trigger the optimization. We plan to leverage constraint solvers [8] to obtain valid test inputs for such tests in future work.

4 SHADOWING OPTIMIZATIONS

Java JIT compilers contain a large number of peephole optimizations. The maintenance becomes difficult as new optimizations are included. When developers want to add a new optimization, they have to be careful that this optimization's effect is not overridden by some existing optimization. Consider two optimizations X and Y in an optimization pass, which are sequentially placed, i.e., X followed by Y. If the set of instructions that Y matches is a subset of the set of instructions that X matches, then Y will never be invoked because X is always invoked before Y for any matched instructions. In this case, we say X *shadows* Y or Y is *shadowed* by X. For example,

```
1: Input: $X$, $Y$: Pattern
2: Output: $res \in \{YES, NO, UNKNOWN\}$ if $X$ shadows $Y$
3: function DETERMINE($X$, $Y$)
4:     $B^x \leftarrow$ before($X$)
5:     $B^y \leftarrow$ before($Y$)
6:     if not SAMESHAPE($B^x$, $B^y$) then
7:         return $NO$
8:     Define a recursive data type $T$ with two constructors:
           nil : int $\rightarrow T$
           tree : opcode $T\ T \rightarrow T$
9:     $V^x, M^x \leftarrow$ CREATENEWVARIABLES($B^x$, $T$)
10:    $V^y, M^y \leftarrow$ CREATENEWVARIABLES($B^y$, $T$)
11:    $\Phi^x \leftarrow$ CONSTRAINSHAPE($B^x$, $M^x$)
12:    $\Phi^y \leftarrow$ CONSTRAINSHAPE($B^y$, $M^y$)
13:    $\Psi \leftarrow$ CONSTRAINEQUIVALENCE($B^x$, $B^y$, $M^x$, $M^y$)
14:    $F \leftarrow \forall_{v^y \in V^y}.\ \Phi^y \Rightarrow \exists_{v^x \in V^x}.\ \Phi^x \wedge \Psi$
15:    return PROVE($F$)
```

Figure 6: Shadow determining algorithm.

Figure 5 shows such a pair of optimizations written in patterns, where pattern ADD2 shadows pattern ADD7.

The shadow problem between two arbitrary optimizations written in patterns X and Y can be rewritten as: for any expression matched by Y, does X match the expression. JOG encodes the problem into an SMT formula and solves it using an SMT solver (Z3[8]). Figure 6 shows the overall algorithm, which we explain using a running example in Figure 5.

The algorithm first extracts `before`'s eASTs (B^x and B^y) from pattern X and Y, respectively (line 4–5), and then checks if B^x and B^y share the same shape (line 6). In the running example from Figure 5, B^x matches B^y node-by-node, except node b in B^y corresponds to both nodes b and c in B^x. Note that function `SameShape` performs a weak instead of exact matching on node types, which allows a leaf node to match with an internal node because a leaf node may represent an expression as well as a variable. Consider an expression `((e + f) - b) + (c - d)`, pattern `((a - b) + (c - d)`

`=> (a + c) - (b + d)`) can still match the expression if we replace `a` with `e + f`.

If B^x and B^y have different shapes, the algorithm will immediately return *NO* for the final result (line 7), i.e, X does not shadow Y. To have such a rough shape check helps the algorithm more efficiently determine the shadow relation for two totally different patterns, which is common in practice. However, having the same shape does not necessarily mean X shadows Y, i.e., any expression matched by B^y can also be matched by B^x. Consider two patterns U, `a + a => ···`, and V, `a + b => ···`, U and V share the same shape but U does not shadow V. A counterexample is expression $1 + 2$ which is matched by V but not U.

To further solve the shadow problem, we describe it formally as: for all expression E_y matched by Y, can we always construct another expression E_x matched by X and ensure that the two expressions are equivalent? If the answer is yes, then X shadows Y; otherwise X does not shadow Y. Note that we say two expressions are equivalent iff they have exactly the same eAST. We make such definition because JIT checks the structure of an expression, rather than evaluate the expression, to determine if an optimization can be applied on the expression. If two expressions are equivalent, they are evaluated to the same value, but the converse does not hold. For example, expression `a + b` and `a + (b + 0)` are always evaluated to the same value but they are not equivalent in our definition. Thus, with this definition of equivalence, the target SMT formula we want to construct is:

$$\forall E_y.\ \big(Y \text{ matches } E_y\big) \Rightarrow \exists E_x.\ \big(X \text{ matches } E_x\big) \wedge \big(E_x = E_y\big).$$

First we construct the formulas for $Y(X)$ matching $E_y(E_x)$. We need to encode B^y into a list of constraints that E_y needs to satisfy in order to be matched. We define a recursive data type T with two constructors: (1) terminal constructor `nil` with no argument, and (2) non-terminal constructor `tree` with the opcode and all the operands as arguments (line 8). We also create a variable with type T for every node in the eAST (line 9 and 10). In our example, the nodes in eAST B^x are represented by variables x_1 to x_7. Next, we encode the shape of the eAST into several constraints (line 11 and 12). For example, the root node of B^x and its two children in Figure 5 satisfies the constraint $x_1 = \text{tree } (+)\ x_2\ x_5$, where x_1, x_2, x_5 is the variable mapped to the root node, the left child, and the right child, respectively. We traverse the entire eAST to add one such constraint for every internal node. Specifically, for a node that represents a constant or number literals, we include an extra constraint on the value contained using the terminal constructor `nil`. Figure 5 lists the constraints encoded from eAST B^x and B^y, resulting in Φ^x and Φ^y, respectively.

Next, we encode equivalence between B^x and B^y into formulas (line 13). We perform a DFS on both eASTs at the same time and add one equivalence relation, in terms of the variables of type T mapped, for every pair of nodes visited, e.g., that the root nodes of B^x and B^y are equivalent is encoded into $x_1 = y_1$. For our running example, every dashed line in Figure 5 connects two equivalent nodes, and Ψ conjoins all the equivalence constraints between B^x and B^y.

With all the constraints ready, we now assemble them into the complete SMT formula (line 14). The final formula F for our running example is

$$\forall y_1, y_2, y_3, y_4, y_5, y_6.\ \Phi^y \Rightarrow \exists x_1, x_2, x_3, x_4, x_5, x_6, x_7.\ \Phi^x \wedge \Psi,$$

where Φ^x is

$$(x_1 = \text{tree } (+)\ x_2\ x_5) \wedge (x_2 = \text{tree } (+)\ x_3\ x_4) \wedge (x_5 = \text{tree } (+)\ x_6\ x_7),$$

Φ^y is

$$(y_1 = \text{tree } (+)\ y_2\ y_5) \wedge (y_2 = \text{tree } (+)\ y_3\ y_4) \wedge (y_5 = \text{tree } (+)\ y_4\ x_6),$$

and Ψ is $(x_1 = y_1) \wedge (x_2 = y_2) \wedge (x_3 = y_3) \wedge (x_4 = y_4) \wedge (x_5 = y_5) \wedge (x_6 = y_4) \wedge (x_7 = y_6)$. Proving the formula then answers whether X shadows Y (line 15). If the formula is valid, then function `Prove` returns *YES*, i.e., X shadows Y; if the formula is not valid, then `Prove` returns *NO*, i.e., X does not shadow Y; if the SMT solver is not able to determine the outcome before timeout, then `Prove` returns *UNKNOWN*. In our example, for any set of y variables, there always exist a set of x variables such that the entire formula holds, e.g., $x_1 \leftarrow y_1$, $x_2 \leftarrow y_2$, $x_3 \leftarrow y_3$, $x_4 \leftarrow y_4$, $x_5 \leftarrow y_5$, $x_6 \leftarrow y_4$, $x_7 \leftarrow y_6$. Thus, the formula F is proven, so *YES* is returned, i.e., X shadows Y.

Neither X nor Y has any preconditions in the example from Figure 5, but JOG can solve the shadow problem for patterns with preconditions. We encode eASTs of preconditions in the same way as eASTs of `before`. Preconditions may contain equivalence on values as well as shapes (e.g., a pattern to match `0 - (x + C)` with precondition `C != 0` where `C` is a constant [40]), so we introduce another set of variables to encode constraints on values. Then we encode both shape and value constraints, and both shape and value equivalence. We construct a target SMT formula involved with both shape and value constraints and equivalence.

5 EVALUATION

We describe the setup of our experiments, quantify code complexity of patterns written using JOG, show performance comparison with hand-written optimizations, and describe test generation and our contributions to OpenJDK.

5.1 Setup

Table 1 is the summary of our work to write 162 patterns using JOG. For the first category of patterns from OpenJDK, we selected `addnode.cpp`, `subnode.cpp` and `mulnode.cpp` in `src/hotspot/-share/opto/` and we studied `Ideal` methods defined in these files. The `Ideal` method reshapes the IR graph rooted at `this` node and returns the reshaped node as an optimized node. Every `Ideal` method may contain many peephole optimizations. We identified and rewrote 68 optimizations into patterns using JOG. For the second category of patterns, we studied LLVM's InstCombine pass that performs numerous algebraic simplifications that improve efficiency, referring to Alive's approach [19]. We translated 92 patterns from `InstCombineAddSub.cpp`, `InstCombineAndOrXor.cpp` in `llvm/lib/Tra-nsforms/InstCombine/`. When we studied optimizations from source code files in OpenJDK and LLVM, we followed the order in which optimizations appear in the files to rewrite them in patterns using JOG, such that the generated C/C++ code from these patterns will be in a proper order. Additionally, we proposed two optimizations and we wrote them as patterns.

Table 1: Summary of patterns that we wrote in JOG.

#Patterns	#OpenJDK	#LLVM	#Original	#PRs
162	68	92	2	7

```
     1       2       3     4     5   6     7   8      9   10  11
long a, long b, long c { before((a - b) + (c - a)); after(c - b); }
```

(a) Pattern code written using JOG (Figure 2a).

```
  1    2    3        4    5    6
Node* in1 = in(1); Node* in2 = in(2);
     7      8     9            10    11    12
int op1  = in1->Opcode(); int op2 = in2->Opcode();
      13       14           15       16       17   18       19   20
  if (op1 == Op_SubL) { if (op2 == Op_SubL && in1->in(1) == in2->in(2) {
                  21    22   23     24   25
    return new SubLNode(in2->in(1), in1->in(2)); } } }
```

(b) Hand-written code in OpenJDK (Figure 2b).

Figure 7: Example of identifier counting.

We ran all experiments on a 64-bit Ubuntu 18.04.1 desktop with an Intel(R) Core(TM) i7-8700 CPU @3.20GHz and 64GB RAM. The SHA of OpenJDK repository [42] we used is `b334d96` and the SHA of LLVM repository [18] is `103e1d9`.

We evaluate JOG by answering the following research questions:

RQ1: How does JOG compare to hand-written optimizations in terms of code complexity?

RQ2: How does the code generated from JOG compare in performance to existing hand-written code in OpenJDK?

RQ3: How effective is JOG at detecting shadows between patterns?

RQ4: How is JOG used to generate tests from patterns and how does it contribute to OpenJDK?

We address RQ1 as to better understand the benefit of using JOG to write optimizations for Java JIT compilers; we use code reduction (in terms of the number of characters and identifiers) as a proxy when answering this question. We address RQ2 to understand the performance of JOG's generated code compared to hand-written code; namely, we wanted to understand the impact on the effectiveness of JIT optimizations. We address RQ3 to study the effectiveness of JOG for detecting shadowing optimizations. We address RQ4 to evaluate JOG's test generation from patterns and describe pull requests we opened for OpenJDK.

5.2 Code Complexity

We select two code features, number of characters and number of identifiers, as a metric to quantify code complexity of patterns written using JOG [6]. We count the number of characters and the number of identifiers for every pattern written using JOG and its counterpart hand-written in OpenJDK. We exclude any white-spaces or newlines when counting characters. We exclude any reserved words in Java or C/C++ languages when counting identifiers, and Figure 7 illustrates the way we count the identifiers using an earlier example (see section 2). Table 2 compares these numbers between hand-written C/C++ code in OpenJDK and corresponding patterns written using JOG. Namely, "Hand-written" means the hand-written C/C++ code in OpenJDK, and "JOG Pattern" means Java code in JOG. The columns of "Reduction (%)" show the percentage of characters and identifiers reduction from hand-written C/C++ code to JOG patterns. Additionally, we provide characters and identifiers of generated C/C++ code from JOG as a reference, which is shown in the columns of "JOG Generated".

Using JOG to write patterns instead of directly writing C/C++ code, the total characters written is decreased from 11,000 to 3,987, and the total identifiers written is decreased from 1,462 to 692. The characters of hand-written C/C++ code is an underestimate of actual numbers because in most cases we do not include the additional lines for declarations of variables due to inconvenience of counting. Due to the same reason, the identifiers of hand-written C/C++ code shown in Table 2 is also an underestimate of actual numbers. However, using JOG to write patterns still shows a significant 63.75% savings in terms of the number of characters and 52.67% savings in terms of the number of identifiers.

There are few groups of optimizations where we write more characters and/or identifiers of code to express them in patterns using JOG. For example, LSHIFT2 is an optimization that transforms `(x >> C0) << C0` into `x & -(1 << C0)`, and LSHIFT3 is a very similar optimization that transforms `(x >>> C0) << C0` into the same result. In OpenJDK these two optimizations are implemented together by including both >> and >>> operators, but we write two separate patterns using JOG, in more lines of code. However, JOG still saves 27.88% characters. We leave how to express the same simplification as in OpenJDK using JOG as future work.

We also count the number of characters and identifiers of generated C/C++ code from JOG (see columns of "JOG Generated"). It is unsurprising that the generated code has much higher numbers of characters and identifiers than hand-written code, because JOG's design of C/C++ code generation prefers consistency to flexibility of coding style, which will benefit future maintenance. For example, `is_int()` and `isa_int()` are used interchangeably in OpenJDK to check if a type is of `int`, but JOG sticks to `isa_int()`, which is recommended, because it returns `NULL` instead of throwing an assertion failure when the checked type is not `int`. After all, as long as generated code keeps the effectiveness of optimizations, it is always preferred to increase maintainability. We will compare performance of generated code and hand-written code in Section 5.3.

5.3 Performance

Our objective with RQ2 is to demonstrate that the performance of JIT does not substantially change when replacing the hand-written code in OpenJDK with code generated from patterns.

A total of 68 optimizations in OpenJDK are replaced using code generated from JOG. To evaluate their performance, we use the *Renaissance* benchmark suite [43] which is a benchmark suite for JVM consisting of 27 individual benchmarks. Some of these benchmarks (*neo4j-analytics*) are incompatible with Java 18 (Java version used in this paper) and are discarded from the experiment. Furthermore, some benchmarks exhibit large variance in their execution times across multiple runs. Since such large variances can lead to inaccuracies in performance evaluation, these benchmarks need to be

Table 2: Comparison of number of characters and number of identifiers between hand-written C/C++ code in OpenJDK and corresponding patterns written using JOG.

Names	Hand-written		JOG Generated		JOG Pattern		Reduction (%)	
	#Chars	#Ids	#Chars	#Ids	#Chars	#Ids	#Chars	#Ids
SUB1	167	24	171	21	47	9	71.86	62.50
SUB2	264	38	412	43	88	17	66.67	55.26
SUB3	340	48	421	45	89	17	73.82	64.58
SUB4	134	20	280	29	38	8	71.64	60.00
SUB5	134	20	280	29	38	8	71.64	60.00
SUB6	134	20	280	29	38	8	71.64	60.00
SUB7	200	31	333	36	48	9	76.00	70.97
SUB8	213	32	395	44	66	10	69.01	68.75
SUB9	169	27	443	45	47	11	72.19	59.26
SUB10	169	27	443	45	47	11	72.19	59.26
SUB11	169	27	443	45	47	11	72.19	59.26
SUB12	169	27	443	45	47	11	72.19	59.26
SUB13	179	26	319	32	71	15	60.34	42.31
SUB14 SUB15 SUB16 SUB17	611	87	1,942	196	188	48	69.23	44.83
SUB18	338	48	344	36	38	5	88.76	89.58
ADD1	284	40	443	50	83	13	70.77	67.50
ADD2	290	38	503	51	60	14	79.31	63.16
ADD3 ADD4	173	25	888	90	94	22	45.66	12.00
ADD5 ADD6	173	25	888	90	94	22	45.66	12.00
ADD7	173	25	443	45	47	11	72.83	56.00
ADD8	173	25	443	45	47	11	72.83	56.00
ADD9	141	21	290	31	38	8	73.05	61.90
ADD10	141	21	290	31	38	8	73.05	61.90
ADD11	462	67	563	65	115	22	75.11	67.16
ADD12 ADD13 ADD14 ADD15	609	85	1,942	196	188	48	69.13	43.53
ADD16 ADD17	710	78	1,530	152	416	44	41.41	43.59
ADD18	204	29	338	37	34	5	83.33	82.76
OR1 OR3	360	38	1,594	157	371	44	-3.06	-15.79
OR2 OR4	351	38	1,598	157	375	44	-6.84	-15.79
MIN1	160	25	292	30	83	19	48.12	24.00
AND1	182	25	327	36	33	5	81.87	80.00
LSHIFT1	326	38	490	54	106	15	67.48	60.53
LSHIFT2 LSHIFT3	165	19	743	80	119	20	27.88	-5.26
LSHIFT4 LSHIFT5	279	34	1,059	104	139	26	50.18	23.53
LSHIFT6	264	28	484	52	92	14	65.15	50.00
RSHIFT1	353	42	479	53	97	14	72.52	66.67
URSHIFT1	303	32	470	55	90	17	70.30	46.88
URSHIFT2	455	55	677	69	90	16	80.22	70.91
URSHIFT3	339	41	472	53	83	14	75.52	65.85
URSHIFT4	246	27	412	44	64	10	73.98	62.96
URSHIFT5	294	39	334	34	54	8	81.63	79.49
$\sum$	11,000	1,462	24,941	2,581	3,987	692	63.75	52.67

discarded. To identify these benchmarks, we built a "vanilla" version of OpenJDK termed as baseline. Each benchmark is executed on the baseline build of OpenJDK five times with each execution consisting of 100 iterations to warm up the JVM and consequently trigger the JIT optimizations. We then compute the coefficient of variance (CV) [4] for each benchmark across the five runs by only considering the last 10 iterations of each run, when JVM is fully warmed up. Benchmarks with CV exceeding 10% are discarded from the experiment. Following these two filtering stages, we excluded 17 more benchmarks and the remaining suitable benchmarks used in the experiment are *log-regression, als, page-rank, finagle-http, scala-kmeans, fj-kmeans, gauss-mix, par-mnemonics* and *dec-tree*.

We now describe the approach used to evaluate the performance of an optimization. The previously identified benchmarks are executed 5 times each on the baseline build with each run once again consisting of 100 iterations. The baseline execution time for a benchmark is then computed by averaging the last 10 iterations over the 5 runs. This procedure is then repeated for each optimization, by replacing the hand-written code in the baseline source with the JOG generated code for the corresponding optimization, to yield the execution time of the benchmarks for that optimization build. A relative difference measure as shown below is then used to evaluate the performance of an optimization (JOG generated) relative to baseline (hand-written):

$$\left.\frac{\text{time}_{\text{hand-written}} - \text{time}_{\text{generated}}}{\text{time}_{\text{hand-written}}}\right|_{\text{benchmark}}$$

where $\text{time}_{\text{hand-written}}/\text{time}_{\text{generated}}$ is the average execution time of a *benchmark* based on bootstrap re-sampling [9] from 50 total executions, i.e., last 10 iterations over 5 runs, of hand-written/generated code. Figure 8 shows the percentage speedup of JOG generated code relative to hand-written code for every group of optimizations in Table 2 and the filtered subset of benchmarks. A positive speedup of generated code relative to hand-written code is marked with an up arrow and a negative speedup (slowdown) with a down arrow. Based on the results of significant difference testing using bootstrap re-sampling, those without statistically significant difference ($p = 0.05$) between JOG generated and hand-written are marked with circles. From Figure 8, most of benchmarks show no significant difference or small differences within the range of 5%. Some benchmarks, e.g., *fj-kmeans* and *gauss-mix*, show more differences. We investigated the benchmarks and found such differences even existed when comparing results of baselines between two experiments, which indicates such benchmarks are more sensitive to noise. Overall, the execution times of OpenJDK build for the *Renaissance* suite with JOG generated code is comparable in performance to that with hand-written code.

5.4 Shadow between Patterns

Recall that JOG is able to check if one pattern shadows another pattern (Section 4). We check each pair of patterns (total 162 patterns) to evaluate JOG's effectiveness on detecting shadows between patterns. We set a timeout with 2 seconds for every check of two patterns and all checks finish within the given time and return definite results (*YES* or *NO*, without *UNKNOWN* as defined in Figure 6). Table 3 enumerates all 9 pairs of patterns where one shadows the other (JOG returned *YES*). In every row, "Shadowing" pattern shadows "Shadowed" pattern. The column of "Before" shows the expression in `before` API, which is the expression to be matched in the

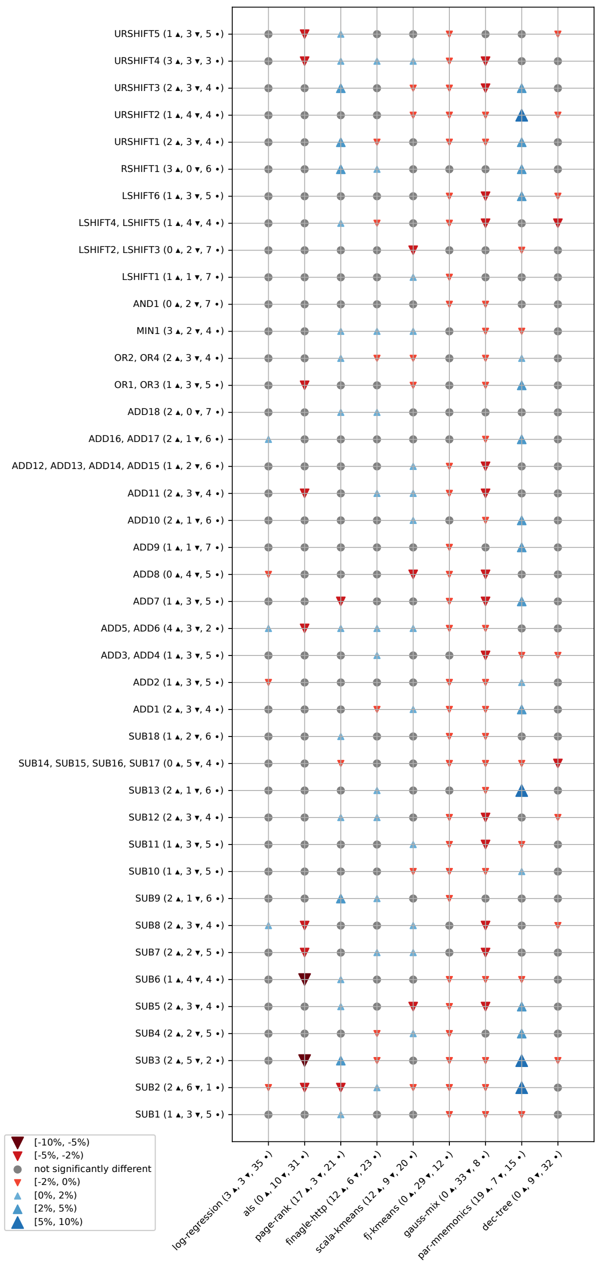


Figure 8: Performance comparison of generated code relative to hand-written code of OpenJDK optimizations on *Renaissance* benchmarks.

pattern, and the column of "Precondition" shows the precondition of the pattern, where ⊤ means no precondition.

In order to see if the reported shadow causes any real world issue, we then manually inspected if any shadowed pattern is placed after shadowing pattern (in execution order) in OpenJDK if both patterns are implemented in OpenJDK, because in this case the shadowed pattern would be entirely shadowed and thus never be reached in the optimization pass (see Section 4). We found that pattern `(a - b) + (c - d)` shadows both pattern `(a - b) + (c - a)` and pattern `(0 - a) + (0 - b)` while both shadowed patterns are put after the shadowing patterns in OpenJDK. We reported this to OpenJDK developers and they confirmed this issue and accepted our pull request to reorder the patterns. We discuss more details for the pull requests in the next section.

Table 3: Shadow between patterns. Constants are in upper-case and free variables are in lower-case.

Shadowing		Shadowed	
Before	Precondition	Before	Precondition

`x - (y + C0)`	`okToConvert(`
	`y + C0, x)`
		`C0 - (x + C1)`	`okToConvert(`
			`x + C1, C0)`

`(a - b) + (c - d)`	⊤	`(a - b) + (b - c)`
		`(a - b) + (c - a)`
		`(0 - a) + (0 - b)`
			⊤
			⊤
			⊤

`x + (0 - y)`	⊤	`(0 - a) + (0 - b)`	⊤
`(0 - y) + x`	⊤	`(0 - a) + (0 - b)`	⊤

`(x + CON) + y`	⊤	`(x + CON1) + CON2`
		`(((z | C2) ^ C1) + 1) + rhs`
		`(((z & C2) ^ C1) + 1) + rhs`
		`(x + 1) + (y ^ -1)`
		`(a + C1) + (C2 - b)`
			⊤
			`C2 == ~C1`
			`C2 == C1`
			⊤
			⊤

`x + (y + CON)`	⊤	`(y ^ -1) + (x + 1)`	⊤
`(x ^ -1) + C`	⊤	`(x ^ -1) + 1`	⊤

`x + (CON - y)`	⊤	`x + (0 - y)`
		`(0 - a) + (0 - b)`
		`(a + C1) + (C2 - b)`
			⊤
			⊤
			⊤

`(CON - y) + x`	⊤	`(CON1 - x) + CON2`
		`(0 - y) + x`
		`(0 - a) + (0 - b)`
			⊤
			⊤
			⊤

5.5 Test Generation & Pull Requests

We use JOG to generate IR tests for all 68 patterns adapted from existing optimizations in OpenJDK (so that we can run those tests with OpenJDK). Excluding the patterns with preconditions for which JOG does not support generating tests, we successfully generate 45 tests. We also generate 8 test classes each of which wraps all the tests for patterns with the same operator, e.g., class `TestSubNode` includes `testSUB1`, `testSUB2`, etc. Next we put the test classes in `test/hotspot/jtreg/compiler/c2/irTests/`, and build and run the tests with OpenJDK. All the 45 tests pass. During our testing, we found 10 tests were missing in OpenJDK; in other words the corresponding optimizations were not tested in OpenJDK. Thus we opened one pull request [38] to add those generated 10 tests to existing test suites of OpenJDK. This pull request has been integrated into the master branch of OpenJDK.

We opened seven more pull requests for OpenJDK, so far. The first category of six pull requests was introducing new JIT optimizations. Figure 9a–9g shows the JOG patterns for new optimizations that we contributed as PRs; we contributed the C/C++ code generated by JOG (and not the patterns). Also, we contributed the IR tests generated by JOG for the patterns. Note that one pull request could contain more than one pattern/optimization. We adapted pattern SUB30 (Figure 9a) from LLVM. We then added generated C/C++ code from the pattern to `SubNode::Ideal` method of OpenJDK, and opened a pull request [31] for the changes. Similarly, we opened another three pull requests [32, 34, 37] that added generated C/C++ code from pattern ADD30 (Figure 9b), pattern ADD31 (Figure 9c), pattern ADD41 (Figure 9d) and pattern ADD42 (Figure 9e) to `AddNode::Ideal` method. Note pattern ADD41 and pattern ADD42 are included in a single pull request. We also opened one pull request [35] for pattern SUB23 (Figure 9f). All the five pull requests have been integrated into the master branch of OpenJDK. We opened one pull request [36] for pattern SUB24 (Figure 9g) and this pull request is under review.

The second category of pull requests we opened was reordering existing JIT optimizations. Figure 9h–9j shows the associated optimizations as patterns. As we mentioned in Section 5.4, pattern ADD2 shadows both pattern ADD7 and pattern ADD8, which means any expression matched by either of pattern ADD7 or pattern ADD8 must be matched by pattern ADD2. Meanwhile in `AddNode::Ideal` method the C/C++ code implementing pattern ADD7 and ADD8 are located after the code for pattern ADD2 (in execution order). Therefore pattern ADD7 or ADD8 would not be reached unless they were moved before pattern ADD2. We reported this issue to OpenJDK developers and initially proposed reordering in our pull request [33]. OpenJDK developers confirmed the issue and then they realized the effects of the two patterns have been done by applying two other optimizations sequentially and thus it is no longer necessary to have these two patterns. Pattern ADD7 and pattern ADD8 were removed from OpenJDK when the pull request was integrated. As future work, we plan to extend our shadow determining algorithm to detect duplicate optimization sequences [12].

6 LIMITATIONS

Internal validity. Our experiments on comparing performance of generated and hand-written optimizations may suffer the threat from noise. To mitigate the threat, we did five end-to-end runs and selected only last 10 fully warmed-up repetitions for measurement. We filtered stable benchmarks by coefficient of variance and we did significant difference test using bootstrap re-sampling for time from all 50 repetitions measured. Although we tried to find the numbers of repetitions and end-to-end runs which are large enough to minimize noise but remains practical for our experiments, choosing different numbers could impact the experimental results.

Construct validity. We used the number of characters and number of identifiers as a metric to quantify code complexity of patterns written using JOG compared to hand-written implementation of optimizations. This metric may or may not reflect complexity for every developer and thus may impact our conclusion on code complexity of optimizations written using JOG.

External validity. We used only *Renaissance* benchmark suites to evaluate performance of optimizations. Although *Renaissance* is the state-of-the-art benchmarks for JVM, to the best of our knowledge, it may still not reflect all use cases of optimizations in real world. Also, the patterns we wrote for evaluation cannot cover all the peephole optimizations. Despite our efforts to increase variety of patterns, such as different compilers, different operations, etc., we cannot ensure the results can be generalized to all peephole optimizations. Last, JOG is designed and developed for Java JIT peephole optimizations in OpenJDK (HotSpot). Thus, JOG will require major changes to be directly generalized to other implementations of Java JIT, e.g., OpenJ9, or other compilers, e.g., LLVM. However, the proposed algorithm of detecting shadow relations between optimizations can be easily implemented for other compilers.

Implementation. Our current implementation of JOG does not generate tests for the patterns with preconditions that specify invariants between variables. For example, one of the patterns written from OpenJDK that transforms `(x >>> C0) + C1` to `(x + (C1 << C0)) >>> C0)` requires a precondition `C0 < 5 && -5 < C1 && C1 < 0 && x >= -(y << C0)` [39]. We plan to leverage constraint solvers to obtain valid test inputs for such tests in our future work.

7 RELATED WORK

We describe related work on (1) DSLs for optimizations, (2) relation between optimizations, (3) finding new optimizations and (4) benchmarking Java JIT.

DSLs for optimizations. A notable area of research addressing the ease of implementing compiler optimizations is in the application of domain specific languages (DSL) for specifying peephole optimizations. One of the first projects [48] introduced a DSL called *Gospel* for specifying compiler optimizations. Cobalt [14] and Rhodium [15] are frameworks to specify peephole optimizations and dataflow analyses, and PEC [13] extends to support loop optimizations. More recently, GCC's *Match and Simplify* [45] introduces a DSL to write expression simplifications from which code targeting GIMPLE and GENERIC is auto-generated. Similarly, Alive [19] is a DSL that can be used for specifying peephole optimizations targeting LLVM. Alive can also be used to generate C/C++ code that can be directly included into LLVM's optimization passes. CompCert [16] is a formalized and verified C compiler in Coq. There is research on verifying SSA-based middle-end optimizer [2], peephole optimizations [25], polyhedral model-based optimizations [7]. Both Alive-FP [23] and LifeJacket [27] prove correctness of floating-point optimizations. The aforementioned tools are designed to verify correctness of optimizations over intermediate representation and introduced DSLs work on the intermediate representation level, while JOG focuses on developers productivity and allows developers to write optimizations in a high-level language (Java), using the existing approach that tests for optimizations are written. Although JOG does not verify optimizations as aforementioned tools, JOG presents an approach to detecting shadow relations between optimizations and JOG can generate IR tests from optimizations specified in patterns.

There has been research that does not focus on verifying optimizations. CAnDL [10] is a DSL for LLVM analysis and it supports use cases beyond peephole optimizations, such as control flows. However, CAnDL requires developers to write constraints to specify optimizations, which is more complicated than high-level expressions JOG takes; also the generated compiler pass is independent to existing code structure in LLVM and thus more difficult to be integrated. COpt [47] is a high-level DSL that allows compiler developers to specify a set of ten high-level optimizations. In contrast to JOG that applies to peephole optimizations, COpt applies on high-level optimizations such as global value numbering, common subexpression elimination, function inlining, etc.

```
@Pattern
public void SUB30(int x, @Constant int c0, @Constant int c1) {
  before(c0 - (x + c1));
  if (Lib.okToConvert(x + c1, c0)) {
    after((c0 - c1) - x); } }
```

(a) Pattern SUB30.

```
@Pattern
public void ADD30(int x) {
  before(x + x);
  after(x << 1); }
```

(b) Pattern ADD30.

```
@Pattern
public void ADD31(int x,
         @Constant int c) {
  before((x ^ -1) + c);
  after((c - 1) - x); }
```

(c) Pattern ADD31.

```
@Pattern
public void ADD41(int x, int y,
         @Constant int con) {
  before(x + (con - y));
  after((x - y) + con); }
```

(d) Pattern ADD41.

```
@Pattern
public void ADD42(int x, int y,
         @Constant int con) {
  before((con - y) + x);
  after((x - y) + con); }
```

(e) Pattern ADD42.

```
@Pattern
public void SUB23(int x,
         @Constant int c) {
  before(c - (x ^ -1));
  after(x + (c + 1)); }
```

(f) Pattern SUB23.

```
@Pattern
public void SUB24(int x,
                    int y) {
  before((x | y) - (x ^ y));
  after(x & y); }
```

(g) Pattern SUB24.

```
@Pattern
public void ADD2(int a, int b, int c, int d) {
  before((a - b) + (c - d));
  after((a + c) - (b + d)); }
```

(h) Pattern ADD2.

```
@Pattern
public void ADD7(int a, int b, int c) {
  before((a - b) + (b - c));
  after(a - c); }
```

(i) Pattern ADD7.

```
@Pattern
public void ADD8(int a, int b, int c) {
  before((a - b) + (c - a));
  after(c - b); }
```

(j) Pattern ADD8.

Figure 9: Associated patterns in pull requests.

Relation between optimizations. There is research [22, 26] on detecting non-termination bugs due to a suite of peephole optimizations applied repeatedly. Termination checking involves determining whether two optimizations can be composited, which is a quantifier-free problem, while the shadow determining problem JOG solves involves with universal quantifiers. Lopes et al. [20] advocated implementation of solver-based tools for finding groups of optimizations that subsume each other to improve existing peephole optimizations. JOG addresses that problem and we plan to explore more relations between optimizations in future work, e.g., duplicate optimizations check.

Finding new optimizations. Optgen [5] exhaustively generates all local optimization rules up to a given cost limit. Barany [1] and Theodoridis et al.'s [46] work compare different compilers' output to find missed optimizations. Unlike them, JOG does not automatically find new optimizations but provides developers a way to easily develop optimizations.

Benchmarking Java JIT. *Renaissance* [43] is recent benchmark suites for JVM, which shows more significant performance differences on evaluating impacts of JIT compiler optimizations than older benchmarks such as DaCapo [4] and SPECjvm2008 [44]; therefore we used *Renaissance* to evaluate performance of generated optimization passes from JOG.

8 CONCLUSION

Writing peephole optimizations requires substantial effort. The current approach of hand-written implementation in Java JIT is not scalable and it is prone to bugs. We presented JOG, a framework that facilitates developing Java JIT peephole optimizations. Compiler developers can write patterns in the same language as the compiler (i.e., Java), using the existing approach for writing tests for peephole optimizations. JOG translates every pattern into C/C++ code that can be integrated as a JIT optimization pass, and generates tests from the pattern as needed. We wrote 162 patterns for optimizations found in OpenJDK, LLVM, as well as some that we designed. Our evaluation shows that JOG reduces the code size and code complexity when compared to hand-written implementation of optimizations while maintaining the effectiveness of optimizations. JOG can also automatically detect possible shadow relations between pairs of optimizations. We utilized this to find a bug in Java JIT as two optimizations could never be triggered as a consequence of being shadowed by another. We opened eight pull requests for OpenJDK, including six on new optimizations, one on removing of shadowed optimizations, and one on new tests of existing untested optimizations, of which seven PRs have been integrated into the master branch of OpenJDK, so far. We believe that JOG will have significant impact in further developments of Java JIT compilers.

ACKNOWLEDGMENTS

We thank Nader Al Awar, Yu Liu, Pengyu Nie, August Shi, Jiyang Zhang, and the anonymous reviewers for their comments and feedback. This work is partially supported by a Google Faculty Research Award, a grant from the Army Research Office, and the US National Science Foundation under Grant Nos. CCF-1652517, CCF-2107291, CCF-2217696.

REFERENCES

[1] Gergö Barany. 2018. Finding Missed Compiler Optimizations by Differential Testing. In *International Conference on Compiler Construction.* ACM, 82–92. https://doi.org/10.1145/3178372.3179521

[2] Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. In *Programming Language Design and Implementation.* Association for Computing Machinery, 4:1–4:35. https://doi.org/10.1145/2579080

[3] Richard Biener and Prathamesh Kulkarni. 2022. *gcc/match.pd at master - gcc-mirror/gcc.* https://github.com/gcc-mirror/gcc/blob/dcb4bd0/gcc/match.pd.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* ACM, 169–190. https://doi.org/10.1145/1167473.1167488

[5] Sebastian Buchwald. 2015. Optgen: A Generator for Local Optimizations. In *International Conference on Compiler Construction*. Springer, Berlin, Heidelberg, 171–189. https://doi.org/10.1007/978-3-662-46663-6_9

[6] Raymond P.L. Buse and Westley R. Weimer. 2010. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering* 36, 4 (2010), 546–558. https://doi.org/10.1109/TSE.2009.70

[7] Nathanaël Courant and Xavier Leroy. 2021. Verified Code Generation for the Polyhedral Model. In *Symposium on Principles of Programming Languages*. ACM, 40:1–40:24. https://doi.org/10.1145/3434321

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[9] Bradley Efron and Robert J. Tibshirani. 1994. *An Introduction to the Bootstrap*. CRC Press. https://books.google.com/books?id=MWC1DwAAQBAJ

[10] Philip Ginsbach, Lewis Crawford, and Michael F. P. O'Boyle. 2018. CAnDL: A Domain Specific Language for Compiler Analysis. In *International Conference on Compiler Construction*. ACM, 151–162. https://doi.org/10.1145/3178372.3179515

[11] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2021. *The Java® Language Specification*. https://docs.oracle.com/javase/specs/jls/se17/jls17.pdf.

[12] He Jiang, Zhide Zhou, Zhilei Ren, Jingxuan Zhang, and Xiaochen Li. 2022. CTOS: Compiler Testing for Optimization Sequences of LLVM. *IEEE Transactions on Software Engineering* 48, 7 (2022), 2339–2358. https://doi.org/10.1109/TSE.2021.3058671

[13] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. In *Programming Language Design and Implementation*. ACM, 327–337. https://doi.org/10.1145/1542476.1542513

[14] Sorin Lerner, Todd Millstein, and Craig Chambers. 2003. Automatically Proving the Correctness of Compiler Optimizations. In *Programming Language Design and Implementation*. ACM, 220–231. https://doi.org/10.1145/781131.781156

[15] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. 2005. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. In *Symposium on Principles of Programming Languages*. ACM, 364–377. https://doi.org/10.1145/1040305.1040335

[16] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[17] LLVM Project. 2022. *llvm-project/llvm/lib/Transforms/InstCombine at main - llvm/llvm-project*. https://github.com/llvm/llvm-project/tree/b26e44e/llvm/lib/Transforms/InstCombine/InstCombineAddSub.cpp.

[18] LLVM Project. 2023. *llvm/llvm-project: The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Note: the repository does not accept github pull requests at this moment. Please submit your patches at http://reviews.llvm.org*. https://github.com/llvm/llvm-project.

[19] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Programming Language Design and Implementation*. ACM, 22–32. https://doi.org/10.1145/2737924.2737965

[20] Nuno P. Lopes and John Regehr. 2018. Future Directions for Optimizing Compilers. https://doi.org/10.48550/ARXIV.1809.02161

[21] W. M. McKeeman. 1965. Peephole Optimization. *Commun. ACM* 8, 7 (1965), 443–444. https://doi.org/10.1145/364995.365000

[22] David Menendez and Santosh Nagarakatte. 2016. Termination-Checking for LLVM Peephole Optimizations. In *International Conference on Software Engineering*. ACM, 191–202. https://doi.org/10.1145/2884781.2884809

[23] David Menendez, Santosh Nagarakatte, and Aarti Gupta. 2016. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *Static Analysis*. Springer, 317–337. https://doi.org/10.1007/978-3-662-53413-7_16

[24] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers.

[25] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified Peephole Optimizations for CompCert. In *Programming Language Design and Implementation*. ACM, 448–461. https://doi.org/10.1145/2908080.2908109

[26] Naoki Nishida and Sarah Winkler. 2018. Loop Detection by Logically Constrained Term Rewriting. In *Verified Software. Theories, Tools, and Experiments*. Springer, 309–321. https://doi.org/10.1007/978-3-030-03592-1_18

[27] Andres Nötzli and Fraser Brown. 2016. LifeJacket: Verifying Precise Floating-Point Optimizations in LLVM. In *International Workshop on State Of the Art in Program Analysis*. ACM, 24–29. https://doi.org/10.1145/2931021.2931024

[28] Oracle and/or its affiliates. 2022. *jdk/AddINodeIdealizationTests.java at master - openjdk/jdk - GitHub*. https://github.com/openjdk/jdk/blob/master/test/hotspot/jtreg/compiler/c2/irTests/AddINodeIdealizationTests.java.

[29] Oracle and/or its affiliates. 2022. *jdk/subnode.cpp at b334d96 - openjdk/jdk - GitHub*. https://github.com/openjdk/jdk/blob/b334d96/src/hotspot/share/opto/subnode.cpp#L163.

[30] Oracle and/or its affiliates. 2022. *jdk/test/hotspot/jtreg/compiler/lib/ir_framework at master - openjdk/jdk - GitHub*. https://github.com/openjdk/jdk/tree/master/test/hotspot/jtreg/compiler/lib/ir_framework.

[31] Oracle and/or its affiliates. 2023. *8277882: New subnode ideal optimization: converting "c0 - (x + c1)" into "(c0 - c1) - x" - Pull Request #6441 - openjdk/jdk*. https://github.com/openjdk/jdk/pull/6441.

[32] Oracle and/or its affiliates. 2023. *8278114: New addnode ideal optimization: converting "x + x" into "x << 1" - Pull Request #6675 - openjdk/jdk*. https://github.com/openjdk/jdk/pull/6675.

[33] Oracle and/or its affiliates. 2023. *8278471: Remove unreached rules in AddNode::IdealIL - Pull Request #6752 - openjdk/jdk*. https://github.com/openjdk/jdk/pull/6752.

[34] Oracle and/or its affiliates. 2023. *8279607: Existing optimization "~x+1" -> "-x" can be generalized to "~x+c" -> "(c-1)-x". - Pull Request #6858 - openjdk/jdk*. https://github.com/openjdk/jdk/pull/6858.

[35] Oracle and/or its affiliates. 2023. *8281453: New optimization: convert ~x into -1-x when ~x is used in an arithmetic expression - Pull Request #7376 - openjdk/jdk*. https://github.com/openjdk/jdk/pull/7376.

[36] Oracle and/or its affiliates. 2023. *8281518: New optimization: convert "(x|y)-(x^y)" into "x&y" - Pull Request #7395 - openjdk/jdk*. https://github.com/openjdk/jdk/pull/7395.

[37] Oracle and/or its affiliates. 2023. *8283094: Add Ideal transformation: x + (con - y) -> (x - y) + con - Pull Request #7795 - openjdk/jdk*. https://github.com/openjdk/jdk/pull/7795.

[38] Oracle and/or its affiliates. 2023. *8297384: Add IR tests for existing idealizations of arithmetic nodes by CptGit - Pull Request #11049 - openjdk/jdk*. https://github.com/openjdk/jdk/pull/11049.

[39] Oracle and/or its affiliates. 2023. *jdk/addnode.cpp at b334d96 - openjdk/jdk - GitHub*. https://github.com/openjdk/jdk/blob/b334d96/src/hotspot/share/opto/addnode.cpp#L358.

[40] Oracle and/or its affiliates. 2023. *jdk/subnode.cpp at b334d96 - openjdk/jdk - GitHub*. https://github.com/openjdk/jdk/blob/b334d96/src/hotspot/share/opto/subnode.cpp#L243.

[41] Oracle Corporation and/or its affiliates. 2023. *JDK Bug System*. https://bugs.openjdk.java.net/browse/JDK-8266601.

[42] Oracle Corporation and/or its affiliates. 2023. *openjdk/jdk: JDK main-line development*. https://github.com/openjdk/jdk.

[43] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Programming Language Design and Implementation*. ACM, 31–47. https://doi.org/10.1145/3314221.3314637

[44] Standard Performance Evaluation Corporation. 2022. *SPECjvm2008*. https://www.spec.org/jvm2008/.

[45] the GCC Developer Community. 2022. *Match and Simplify*. https://gcc.gnu.org/onlinedocs/gccint/match-and-simplify.html.

[46] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding Missed Optimizations through the Lens of Dead Code Elimination. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 697–709. https://doi.org/10.1145/3503222.3507764

[47] Sruthi Venkat and Preet Kanwal. 2018. COpt: A High Level Domain-Specific Language to Generate Compiler Optimizers. In *International Conference on Advanced Computation and Telecommunication*. IEEE, 1–6. https://doi.org/10.1109/ICACAT.2018.8933593

[48] Deborah L. Whitfield and Mary Lou Soffa. 1997. An Approach for Exploring Code Improving Transformations. *ACM Trans. Program. Lang. Syst.* 19, 6 (1997), 1053–1084. https://doi.org/10.1145/267959.267960

Received 2023-02-16; accepted 2023-05-03