ABSTRACT

We present JOG, a framework for developing peephole optimizations and accompanying tests for Java compilers. JOG allows developers to write a peephole optimization as a pattern in Java itself. Such a pattern contains code before and after the desired transformation defined by the peephole optimization, with any necessary preconditions, and the pattern can be written in the same way that tests for the optimization are already written in OpenJDK. JOG automatically translates each pattern into C/C++ code as a JIT optimization pass, and generates tests for the optimization. Also, JOG automatically analyzes the shadow relation between a pair of optimizations where the effect of the shadowed optimization is overridden by the other. We used JOG to write 162 patterns, including many patterns found in OpenJDK and LLVM, as well as some that we proposed. We opened ten pull requests (PRs) for OpenJDK, introducing new optimizations, removing shadowed optimizations, and adding generated tests for optimizations; nine of PRs have already been integrated into the master branch of OpenJDK. The demo video for JOG can be found at https://youtu.be/z2q6dhOiogw.

KEYWORDS

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

ACM Reference Format:

1 INTRODUCTION

Peephole optimizations [11, 13] belong to an essential class of compiler optimizations that examine a few adjacent code instructions or a basic block, known as a window, and make targeted changes to improve performance or reduce the code’s size, e.g., \( A + A \) is transformed into \( A \ll 1 \). Peephole optimizations are widely used in popular compilers such as GCC, LLVM, and Java Just-in-Time compilers (Java JIT for short) [2, 9, 16].

Peephole optimizations are typically implemented as compiler passes, such that each detects a window and replaces it with an optimized form. Implementation of an optimization is commonly done in the language in which the compiler itself is implemented (e.g., C/C++ for Java JIT), using the compiler infrastructure, e.g., internal data structure representation, to manipulate windows. This low-level internal representation is quite different from the actual code (written in Java) being optimized. The mismatch hinders developers from effectively reasoning about windows of interest, because they have to repeatedly map instructions from high-level code (e.g., Java) to low-level code (e.g., C/C++) and data. The mismatch also makes implementation error-prone [7, 8, 19, 24, 26–28].

Alive [10] improves the traditional approach by introducing patterns, which are written in a domain specific language (DSL) and manipulate LLVM bitcode. Developers can write patterns in the DSL which are then translated into compiler passes. However, Alive still remains significantly detached from the programming language it optimizes (C++), leading to a steep learning curve and it lacks support for software tools, e.g., syntax highlighting in IDEs.

Our key insight is that many peephole optimizations can be expressed within the programming language being optimized, thus avoiding complex patterns that manipulate low-level code representations. In OpenJDK, a significant portion of JIT optimization tests (known as IR tests) are written in Java and incorporate specific patterns within their code to trigger the optimizations being evaluated [15]. We propose to extend the concept, not only to use patterns to write IR tests but to comprehensively describe the entire optimization, encompassing both code before and after the optimization, which in turn implicitly describe IR tests.

We present JOG [25], which enables developers to write peephole optimizations for Java JIT as high-level Java statements. These patterns undergo Java compiler type-checking and are automatically translated into compiler passes (in C/C++) by JOG. Furthermore, JOG can automatically generate IR tests in (Java) from these patterns. By writing patterns in Java for Java JIT, we ensure the meaningfulness of statement sequences within programs, i.e., windows, can indeed appear in programs (a guarantee not always achieved when working with IRs or compiler abstractions). Our approach also simplifies the rationale behind each peephole optimization, transforming what was once extensive comments or test cases into self-explanatory patterns. Moreover, developers can leverage software engineering tools like IDEs and linters while creating patterns in JOG. Having patterns written in Java also opens the door for future program equivalence checkers [1] compatible with both Java code and bytecode, readily obtained by compiling JOG patterns.

The brevity of patterns eases the analysis of relations between 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. For instance, 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
Figure 1: An example IR test available in OpenJDK (SHA fd910f7) [18].

- c, with variables a, b, c, and d. Notably, any expression matching 
  \((a - b) + (b - c)\) (Y) also matches \((a - b) + (c - d)\) (X). If 
  X is always applied before Y in a compiler pass, the effect of X will 
  shadow Y. JOG can automatically report this shadow relation.

Using JOG, we wrote 162 optimization patterns: 68 from Open-
JDK, 92 adapted from LLVM, and two entirely new. Most OpenJDK 
patterns were taken from existing tests or hand-written examples in 
C/C++ comments. Our most complex pattern is just 115 characters, 
compared to the 462-character C/C++ counterpart that manipulates 
the IR. Our evaluation confirms that

JOG is open source and publicly available at https://github.com/ 
EngineeringSoftware/jog.

2 EXAMPLE

Figure 1 shows a test written using the IR test framework [17] which 
is a recommended approach to testing JIT peephole optimizations 
in OpenJDK. The test is expected to compile the annotated (@Test) 
method test8 and optimize \((a - b) + (c - a)\) to \(c - b\); the 
expected transformation is written as a comment. The IR shape 
of the compiled method is checked against certain rules specified 
using the @IR annotation (lines 2–3). The rules validate that 
the compiled method does not contain ADD node (line 2) and contains 
exactly one SUB node (line 3).

Using JOG, developers can write an optimization, i.e., \((a - b) + (c - a)\) 
\(\rightarrow\) \((a - b)\) to \(c - b\), in a way that mirrors the existing IR test. 
In Figure 2a, a pattern written in JOG is a Java method annotated 
with @Pattern. The method’s parameters (line 2 in Figure 2a) declare 
variables (a, b, and c), specifying the data type of each as long. 
Inside the method, two API calls, before((a - b) + (c - a)) (line 3 in Figure 2a) and after((c - b) (line 4 in Figure 2a), define the 
expressions before and after the optimization. Both calls follow 
the format of existing IR tests. before((a - b) + (c - a)) 

returns code from the existing test return (a - b) + (c 
- a); (line 6 in Figure 1), and after((c - b) is taken from the 
comment // Check (a - b) + (c - a) => (c - b) (line 4 in 
Figure 1). Moreover, since the pattern and the test follow the same 
structure, not only does JOG enable developers to write patterns, 
but it can also automatically generate IR tests from patterns.

JOG automatically translates a pattern into C/C++ code for direct 
inclusion in a JIT optimization pass (Figure 2b). Figure 2c displays 
hand-written code extracted from OpenJDK, achieving the same 
JIT peephole optimization to transform \((a - b) + (c - a)\) 

into \(c - b\). The implementation matches expressions of interest 
and then returns a new optimized equivalent expression. In this 
example, the matched expression must meet four conditions: (1) It 
is an addition expression (implicitly line 1 in Figure 2b because 
the method belongs to AddlNode); (2) its left operand is a subtraction 
expression \((a - b)\) (line 12 in Figure 2b); (3) its right operand is a 
subtraction expression \((c - a)\) (line 13 in Figure 2b); and (4) both 
subtraction expressions share a same operand \((a)\) (line 14 in 
Figure 2b). Once a match is found, the code constructs the new 
subtraction expression \((c - b)\) using b and c (line 15 in Figure 2b), 
reducing the evaluation cost from two subtractions and one addition 
to a single subtraction. Notably, a bug existed in the OpenJDK 
code due to incorrect access to the right operand of the right sub-
expression (line 8 in Figure 2c), taking 13 years to discover it [19]. 
If JOG had been used for implementing the optimization, this bug 
could have been avoided.

JOG analyzes the before and after API calls to infer conditions 
and construct new expressions, eventually generating C/C++ code 
as compiler passes. Figure 2b shows code generated from the pattern 
in Figure 2a, preserving functionality and avoiding the bug found 
in the hand-written code shown in Figure 2c.
### 3 TECHNIQUE AND IMPLEMENTATION

Figure 3 shows a high-level overview of the workflow of the JOG framework. In this section, we briefly describe the design and implementation of patterns, translation details, test generation, and shadow relation detection [25].

**Design and implementation of patterns.** As the example in Figure 2a shows, we define the syntax of patterns using a subset of the Java programming language, where each optimization is represented as a Java method annotated with @pattern. The parameters of these methods declare variables used in patterns, with two types: constant values (representing literals that are annotated with @Constant) and free variables (representing any expression). We also provide two API methods, void before(int expression), which specifies the expression to match in the pattern, and void after(int expression), which specifies the optimized expression (int can also be long). A valid pattern must contain both a before and after method call in the method body, which may also feature if statements for preconditions and assignments for local variable re-assignments.

**Translation.** JOG translates patterns into C/C++ code that implements compiler passes for JIT optimizations. JOG starts translation with parsing the expression provided in the before API and constructing an extended abstract syntax tree (eAST) for it. The eAST represents the structure of IR that matches the expression, which is essentially a directed acyclic graph (DAG). JOG maps identifiers in the pattern to eAST nodes. The same identifiers are reused to construct eAST for the after API. Figure 4 shows the eASTs constructed from the pattern ADD8 (Figure 2a). Next, JOG creates an if statement where the condition represents the necessary conditions for expression matching. These conditions may check operators, constants, identical identifiers, etc., and any preconditions specified in the pattern. The “then” branch of the if statement ends with a return statement providing the optimized expression. Finally, JOG prepends the if statement with proper variable declarations, concluding translation of the pattern. When handling multiple patterns, JOG follows the order specified in the provided file.

**Test generation.** We use the example in Figure 1 to describe how JOG generates an IR test from the pattern in Figure 2a. The @Test method first declares exactly the same free variables as the pattern (long a, long b, long c), and returns exactly the expression inside the before API in the pattern (return (a - b) + (c - a)). One exception is that when the pattern has a constant variable, 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 creates @IR annotations (@IR(counts = IRNode.SUB, "1") and @IR(failOn = IRNode.ADD)).

**Shadowing optimizations.** 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, e.g., X transforms (a - b) + (c - d) into (a + c) - (b + d), and Y transforms (a - b) + (b - c) into a - c, with variables a, b, c, and d. Given a pair of optimizations expressed in patterns X and Y, JOG rewires the problem of whether X shadows Y formally as follows: For every expression E matched by Y, it is also matched by X? JOG then encodes this problem in an SMT formula and leverages a constraint solver (Z3 [5]) to obtain a result on the shadow relation between the given pair of patterns [25].

### 4 TOOL INSTALLATION AND USAGE

JOG requires JDK 11 or later versions. We describe the installation steps and usage instructions using a Linux system (Ubuntu 20.04) with GNU Bash (version 5.0) as an example. We also provide a docker image that contains a built OpenJDK and the cloned JOG repository, which can be obtained by docker pull zzqut/jog:latest.

**4.1 Installation**

The first step is to clone the JOG repository¹.

```
$ git clone https://github.com/EngineeringSoftware/jog
$ cd jog
```

To install JOG, one can execute the installation script like so:

```
$ ./tool/install.sh
```

¹We provide the icse24-demo tag for the archive purpose.
We wrote 162 patterns using JOG. We submitted a pull request to include these 10 tests in OpenJDK's JDK. We discovered that 10 tests were missing in OpenJDK, indicating the corresponding optimizations had not been tested. Thus, we submitted a pull request to include these 10 tests in OpenJDK's existing test suites. This pull request has been integrated into the master branch of OpenJDK (SHA fd910f7).

4.2 Usage

After installation, one can run JOG through the executable jar ./tool/jog.jar. We provide an example file Example.java in the repository, which contains two patterns. To run JOG:

$ java -jar tool/jog.jar Example.java

This command calls a bash script to build the JOG jar. If the command completes normally, an executable jar jog.jar will appear in the tool directory, i.e., ./tool/jog.jar.

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

<table>
<thead>
<tr>
<th>#Patterns</th>
<th>#OpenJDK</th>
<th>#LLVM</th>
<th>#Original</th>
<th>#PRs</th>
</tr>
</thead>
<tbody>
<tr>
<td>162</td>
<td>68</td>
<td>92</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

This command generates C/C++ code as compiler passes, (b) generates IR tests for the optimizations, and (c) reports a shadow relation between the pair of patterns (optimizations) provided. Figure 5 shows a screenshot of running the command. JOG saves the generated C/C++ code in cpp files with names matching the top level operator of the before API, e.g., generated code for pattern ADD2 with before((a-b)+(b-c)) is saved into addnode.cpp. To integrate these compiler passes into OpenJDK, one can simply copy the contents of these cpp files into the corresponding files in OpenJDK with identical names. JOG also generates IR tests as java files, which can be directly run with OpenJDK IR testing framework.

5 EVALUATION

We wrote 162 patterns using JOG, as detailed in Table 1, spanning three categories. First, we studied the Ideal methods from OpenJDK, and we identified and rewrote 68 optimizations into patterns found in addnode.cpp, subnode.cpp and mulnode.cpp within src/hotspot/share/opto/. Second, we studied LLVM's InstCombine pass, inspired by the Alive approach [10], and translated 92 patterns from InstCombineAddSub.cpp and InstCombineAndOrXor.cpp in llvm/lib/Transforms/InstCombine/. Additionally, we proposed two optimizations and wrote them as patterns.

We evaluated the code size and complexity [4] of the 68 patterns rewritten from OpenJDK using JOG. Compared to hand-written optimizations, using JOG to write patterns reduced the total character count from 11,000 to 3,987 (by 63.75%), and the total identifiers from 1,462 to 692 (by 52.67%). We also evaluated JOG-generated C/C++ code performance in comparison to hand-written code as compiler passes using the Renaissance benchmark suite [20]. Overall, we found no significant difference on execution time (which is average over 5 runs) between hand-written code and JOG-generated code.

Furthermore, we used JOG to generate tests from patterns we wrote. We discovered that 10 tests were missing in OpenJDK, indicating the corresponding optimizations had not been tested. Thus, we submitted a pull request to include these 10 tests in OpenJDK's existing test suites. This pull request has been integrated into the master branch of OpenJDK (SHA fd910f7).

6 RELATED WORK

Notable research explores implementing compiler optimizations using domain specific languages (DSLs). While prior works [6, 10, 21, 23] have introduced DSLs operating at the intermediate representation level of GCC or LLVM, JOG takes a different approach: JOG prioritizes developer productivity, allowing optimizations to be written in a high-level language (Java) using an approach very similar to the one for writing tests for optimizations. Also, researchers have explored relations between optimizations, such as detecting non-termination bugs due to repeated application of peephole optimizations [12, 14], and automatic discovery of new optimizations [3, 22].

7 CONCLUSION

Writing peephole optimizations is labor-intensive and error-prone. We introduced JOG, a framework that simplifies development by allowing patterns to be written in Java and then automatically translating them into C/C++ that can be integrated as a JIT optimization pass. JOG can also generate IR optimization tests in Java from the patterns and uncover shadow relations between optimizations. We wrote 162 patterns from OpenJDK, LLVM, along with some we adapted from LLVM or proposed ourselves. One PR is currently under review, and the remaining seven PRs have been accepted and integrated into the master branch of OpenJDK. In the future, we plan to prepare more PRs for the patterns we already wrote.

Table 2: Pull requests we submitted to OpenJDK.

<table>
<thead>
<tr>
<th>Type</th>
<th>Id</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>New optimizations</td>
<td>7395</td>
<td>Submitted</td>
</tr>
<tr>
<td>7396</td>
<td></td>
<td>Merged</td>
</tr>
<tr>
<td>7795</td>
<td></td>
<td>Merged</td>
</tr>
<tr>
<td>16333</td>
<td></td>
<td>Merged</td>
</tr>
<tr>
<td>16334</td>
<td></td>
<td>Merged</td>
</tr>
<tr>
<td>Shadowing optimizations</td>
<td>6752</td>
<td>Merged</td>
</tr>
<tr>
<td>New tests</td>
<td>11049</td>
<td>Merged</td>
</tr>
</tbody>
</table>

In total, we submitted ten pull requests (PRs) to OpenJDK (see Table 2). In addition to the aforementioned PR for missing tests, we identified two shadow relations where one optimization was found to override the effect of another, and we reported the issue through a PR, which have been confirmed and resolved. Also, eight other PRs introduced new JIT optimizations based on patterns we adapted from LLVM or proposed ourselves. One PR is currently under review, and the remaining seven PRs have been accepted and integrated into the master branch of OpenJDK. In the future, we plan to prepare more PRs for the patterns we already wrote.

ACKNOWLEDGMENTS

We thank Nader Al Awar, Yu Liu, Pengyu Nie, August Shi, Fu-Yao Yu, 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-2107291, CCF-2217696, and CCF-2313027.
REFERENCES


