Automatic Migration of Java Platform Threads to Virtual Threads

by

Nipuni Tharushika Hewage

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Language features are inevitable in any programming language. Language maintainers introduce new language features and enhancements to existing features with their releases. For example, Virtual threads emerged as a preview feature with the Java 19 release under project Loom. Developers may need to migrate traditional thread usages with virtual threads. However, migrating to new language features is not easy. It consumes time and resources because it may require a steep learning curve to understand the required migration process. Developers may also be concerned about the after-effects of the migration, mostly on the performance of the application.

Several studies have focused on mining refactoring patterns and automatic migrations in different scenarios such as in the modification of libraries, and frameworks. We found out that those studies do not align with the requirements of Loom migration because we do not have sufficient historical usage in open-source repositories and proper documentation along with source code. In this thesis, we present Loomizer, a transformation tool that is capable of migrating traditional thread usages in a Java application to virtual threads. Loomizer is based on Rascal. We apply Loomizer on various application servers such as Open Liberty, Tomcat, Wildfly, and Undertow.

We demonstrate an empirical evaluation focusing on the performance changes, specifically throughput, on those application servers after migration. We find that the throughput improvement with virtual threads is not always noticeable. In the presence of entirely blocking operations in the application de-

ployed on the servers, we observe throughput improvement and deterioration with different delays in the blocking operations compared to the servers before migration. Additionally, the presence of CPU-consuming operations and blocking operations in the application limits the throughput improvement with virtual threads in all the application servers except in Open Liberty. We discover that throughput improvement with virtual threads also depends on the application server, the type of the application (whether IO-bound or CPU-bound), the expected workload, and the delay in the blocking operations in the application.

Preface

This thesis is an original work by Nipuni Hewage. I was responsible for the implementation of LOOMIZER, experimental setups, and the empirical evaluation in this thesis. Karim Ali was the supervisory author of this work and contributed to technical advice, designing experimental scenarios, and many thesis edits.

Acknowledgements

First of all, I would like to express my gratitude to Prof.Karim Ali, my supervisor, for your patience, motivation, and guidance. Your constructive comments and feedback were always useful and this would not be realistic without your tremendous support.

Thank you to Prof.Sarah Nadi, for your encouragement, and patience, and for being my advisor until I found a supervisor.

Thank you to folks at IBM including Vijay Sundaresan, Gary DeVal, and Daryl Maier for the constructive discussions we had regarding experimental processes and setups. They were informative and we utilized the explained approaches in this work.

Above all, Thank you to my beloved family for their blessings, continuous support, and believing in me.

Thank you University of Alberta, Department of Computing Science for this opportunity to pursue my master's degree. Further, my thanks go to my colleagues at Maple Lab for being supportive and mentoring me whenever necessary.

Contents

List of Tables

List of Figures

Glossary

API
: Application Programming Interface

AST
: Abstract Syntax Tree

DSL
: Domain Specific Language

FIFO
: First In, First Out

JCA
: Java Cryptographic Architecture

JDK
: Java Development Kit

JIT
: Just-In-Time

LOC
: Lines of Code

NMON
: Nigel's performance Monitor for Linux

Chapter 1

Introduction

GitHub has recently ranked Java as the 5th topmost used language for software development [11]. Over the years, Java has offered several new language features such as generics [16], functional interfaces [25], virtual threads [7], and structured concurrency [8]. To address the evolving requirements of developers, language maintainers perform language releases that introduce new language features and enhance existing ones. For example, Oracle has released a new Java version every year since Java 9 [9]. They have also issued release notes [28] that contain information about new language features, removed/deprecated features, and known issues in a particular release. Introduced in Java 19 under project Loom [31], virtual threads are comparatively lightweight threads that enable easily writing and monitoring high-throughput concurrent applications [7]. Applications that follow the thread-per-request model can benefit from virtual threads in terms of higher throughput and scalability [7].

With the introduction of new language features such as Loom, developers may want to upgrade their software applications to use them. This process requires developers to get a good grasp on the new features, which may lead them to go through a steep learning process. Therefore, developers typically use automated tools to perform this migration. The general recipe for this process consists of 3 main steps: mining refactoring patterns, detecting those patterns, and applying code transformations.

To mine refactoring patterns, existing work (e.g., DAAMT [44], SemD-

IFF [10], NEAT [38], PATTERIKA [3]) usually analyzes the source code with specific code changes or proper replacement messages in code documentation or using commit histories in source code repositories. While this approach is effective, we cannot apply it to new language features such as Loom, because such source code differences and replacement messages do not exist yet. Once developers obtain refactoring patterns, they can locate the instances of identified patterns in their codebases using several existing tools [19]–[22]. While TAPIR [20] and JELLY [21] support only Java script, COGNICRYPT [19] operates on Java source code. HOTFIXER [22] is an existing code transformation tool based on COGNICRYPT and SOOT [41], where code transformations happen at the bytecode level. Due to the nature of applying security patches to a program at runtime, HOTFIXER is not suitable to use for source-level code transformations to perform a codebase migration to use Loom.

To address the limitations of existing work, we present LOOMIZER, our tool that is based on RASCAL [18], which can automatically detect and transform platform threads to virtual threads in a given Java source code. RASCAL offers exciting features such as source-to-source transformation [36], pattern matching [29], and support for recent Java versions [34]. Therefore, LOOMIZER enables code transformation at the source level, enabling developers to easily use it for Loom migrations within their development environment. To encode migration patterns in LOOMIZER, we first manually extracted a set of refactoring patterns by studying the JDK Enhancement Proposal for virtual threads [7].

Migrating to Loom migration may impact the properties of software systems because of virtual threads and their benefits. Having a huge runtime overhead from poor performance will lead to unsuccessful applications [6], affecting business revenues. Therefore, a critical factor for developers to assess is the effect of migrating their code to a new language feature on the performance of their software system. Therefore, we have evaluated the effectiveness of LOOMIZER by applying it to several application servers such as Open Liberty [24], Tomcat [39], Wildfly [42], and Undertow [40]. We then validated the preciseness of the applied transformations, as well as measured the effect of

the applied transformations on the performance of those application servers.

The following are the contributions of this thesis.

- We present a set of manually mined refactoring patterns for virtual threads.
- We propose Loomizer to automatically migrate platform threads to virtual threads in a Java application.
- We apply migrations on several application servers using Loomizer. We also manually validate the correctness of the applied transformations.
- We evaluate performance changes in the migrated application servers using several experimental setups.

The remainder of the thesis is organized as follows. Chapter 2 describes the background materials about Loom features (virtual threads) with the limitation identified in platform threads. We explore several prior works and analyze them to validate the feasibility of their use in our migration scenario under Chapter 3. In Chapter 4, we discuss the refactoring patterns that we manually mined, the main components in Loomizer, and how Loomizer performs migrations for each pattern considering data types. We also present a few examples of transformations completed with Loomizer. Chapter 5 contains the experimental setups and results of experiments that we conducted to evaluate performance impacts on migrated application servers. We also present a few statistics, related to the number of transformations and time spent on each server, we recorded with Loomizer applying on application servers. Lastly, Chapter 6 provides the conclusions of our work with future directions.

Chapter 2

Background

This chapter includes information about traditional platform threads and their limitations, as well as Project Loom.

2.1 Platform Threads

Figure 2.1 shows how a Java platform thread attaches to an OS thread with 1:1 mapping. Since each Java program needs at least one thread to execute its main method [12], a platform thread typically runs code on its attached OS thread [7]. To scale to a large number of requests, most applications will follow a thread-per-request model. In that model, a thread can only handle a particular request until it is completely processed [7]. Hence, the platform thread, on which a request runs, captures the attached OS thread for the duration of the request though the request endpoint contains a prolonged wait

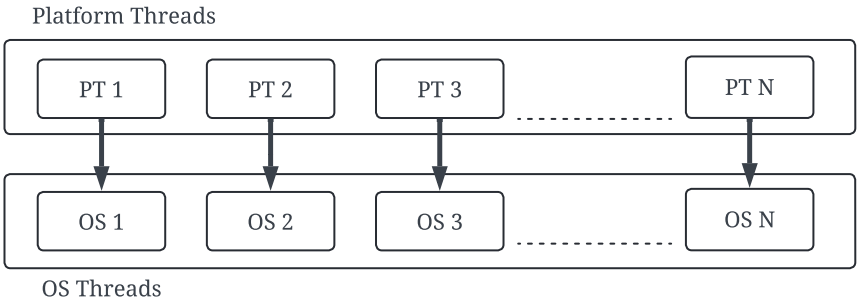

Figure 2.1: The Mapping of Platform Threads and OS Threads in Java.

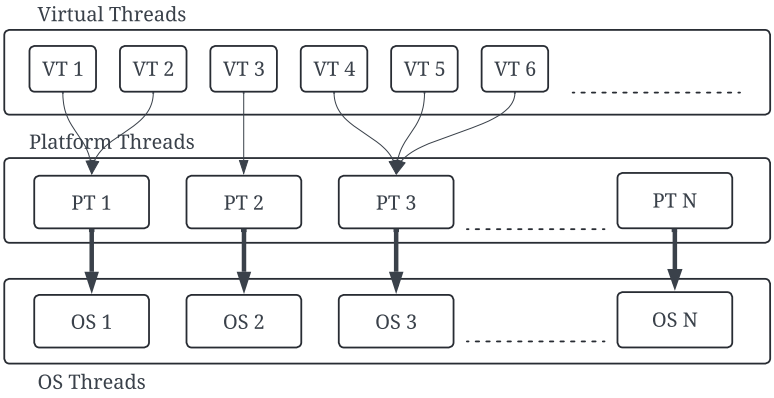

Figure 2.2: Scheduling multiple virtual threads to a single platform thread.

time [7]. Serving a higher number of incoming requests limits the number of requests processed during a given time because each thread has to be idle till the time defined in the request endpoint has elapsed before executing any other requests. Therefore, developers cannot obtain a better throughput if the request contains a high wait time and the application follows the thread-per-request model under a higher load of requests.

If developers want to increase throughput, they have to improve concurrency (i.e., the number of requests processed at the same time). Since each thread has to process only one request at a time, developers have to increase the number of platform threads and, as a result, the number of attached OS threads. However, the number of available OS threads is limited, which causes a barrier to increasing the number of associated platform threads. Therefore, scaling platform threads is limited.

2.2 Project Loom

In 2017, Java language maintainers initiated project Loom with the goal of "drastically reducing the effort of writing, maintaining, and observing high-throughput concurrent applications" [31]. To address the scalability issue of platform threads, Loom provided virtual threads as a preview feature in Java 19. Figure 2.2 shows that, compared to platform threads, virtual threads are lightweight and may not attach to OS threads in 1:1 mapping during the

```
1 Thread thread = Thread.ofVirtual().unstarted(runnable);
2
3 Thread thread = Thread.ofVirtual().name("th01").unstarted(runnable);
```

Figure 2.3: Creation of a Virtual Thread

lifetime of a request. Moreover, there might be many virtual threads for fewer OS threads resembling M:N mapping [7]. There is a scheduler in the Java Development Kit (JDK) to manage and assign virtual threads to platform threads and it utilizes a work-stealing ForkJoin pool algorithm, where the idle threads steal work from busy threads [7], and First In, First Out (FIFO) mode. During the lifespan of a request, the scheduler may assign a virtual thread, on which Java code runs, to different platform threads (i.e., carrier threads) especially when there is a high wait time in the request. Unlike using platform threads, the scheduler unmounts the virtual thread from its attached platform thread during high wait time in the processing request [7]. The scheduler can assign another busy virtual thread to that platform thread for execution because the platform thread is free. Once the high wait time has elapsed, the scheduler assigns the virtual thread back to an available platform thread for processing. With this approach, virtual threads on which the Java code runs do not block processing of any other requests even under a higher number of incoming requests. Therefore, virtual threads should scale better. There are also situations where the scheduler cannot unmount the virtual thread from its carrier thread as it is pinned to the platform thread [7]. Then, virtual threads may not exhibit throughput improvement.

Virtual threads are different from platform threads in several aspects. Virtual threads are comparatively lightweight and inexpensive while platform threads are costly because the operating system has to allocate a large amount of memory (MB) during thread initialization. Hence, developers should not create pools of virtual threads [7]. Moreover, thread groups do not have a meaning with virtual threads [7].

Loom exposes a set of APIs to create and use virtual threads. Figure 2.3 shows how to create a virtual thread with and without a name. Developers

```
4 Thread thread = Thread.ofVirtual().start(runnable);
5
6 Thread thread = Thread.startVirtualThread(runnable);
```

Figure 2.4: Creating and Starting a Virtual Thread

```
7 ThreadFactory threadFactory = Thread.ofVirtual().factory();
8 Executors.newThreadPerTaskExecutor(threadFactory);
9
10 Executors.newVirtualThreadPerTaskExecutor();
```

Figure 2.5: Creating Virtual Threads for Each Task

can also create and execute a virtual thread using either statement depicted in Figure 2.4. Figure 2.5 illustrates the creation of `ExecutorService` which can create a thread per task in two ways.

Chapter 3

Related Work

In this chapter, we discuss prior work related to automatically mining code refactoring patterns, and transforming code. We also explain the limitations of existing work when applied to new language feature migration.

3.1 Mining Migration Patterns

Xi et al. [44] have introduced DAAMT (Deprecated API Assisted Migration Tool). To detect instances of deprecated APIs, DAAMT generates refactoring patterns only if there is proper documentation with complete replacement messages. For Loom APIs, we do not have source code with documentation that contains enough information related to API changes. Therefore, this approach is not applicable to migrating existing codebases to use Loom.

When APIs of a framework change, client programs that use that particular framework should also be changed accordingly. SemDiff [10] suggests changes in a client program by analyzing the modifications made to a framework. Since Loom is a completely new framework, it does not have any code changes referring to platform threads. Therefore, we will also not be able to use SemDiff to mine refactoring patterns for Loom.

NEAT (No Example API Transformation) [38] generates refactoring patterns for Android APIs without any code examples. To generate transformation rules, NEAT requires the affected library source code in which both the deprecated API and the replacement API exist with information about both APIs. However, Loom is a new language feature. It does not replace or depre-

cate any previous API related to platform threads. Therefore, NEAT cannot mine refactoring patterns for Loom migrations.

To address the difficulty in manually generating migration patterns, PATTERIKA [3] uses historical commits to mine refactoring patterns utilizing Abstract Syntax Trees (ASTs) and code differencing graphs. Since Loom is still a new language feature, there are not enough historical commits in open-source repositories, as a result, it becomes a barrier for us to use PATTERIKA.

If a developer needs to change the data type of a set of variables (e.g., int → long), they would also need to perform type changes in all their variable references (e.g., method calls). Ketkar et al. [17] have proposed TC-INFER, which extracts refactoring patterns by analyzing version histories of open-source repositories with similar changes. Similar to PATTERIKA, the lack of Loom changes in open-source repositories becomes the obstacle for TC-INFER.

Loom is a new language feature. Hence, we cannot use existing work to mine refactoring patterns for Loom migrations. Therefore, to bootstrap a migration tool for Loom, we need to manually extract a set of refactoring patterns by studying the JDK enhancement proposal [7] for virtual threads.

3.2 Code Transformation

After mining refactoring patterns for Loom, our next objective was automatically detecting and migrating the extracted patterns. Several prior work have focused on code transformation by developing Domain Specific Languages (DSLs) to encode the mined patterns.

Møller et al. [20] present TAPIR, a tool that identifies the impacted locations in a JavaScript code due to library changes. JSFIX [23] is a transformation tool that brings out several semantic patches to complete transformations along with an interactive process where the user has to answer various questions. JSFIX utilizes TAPIR to locate the impacted code. Since TAPIR and JSFIX primarily work for JavaScript programs, we could not use them to transform Java code automatically without human interference.

JELLY [21] is a static analyzer for pattern matching on JavaScript and

TypeScript programs. It uses Babel [1] to parse JavaScript code into an AST. We would have had to re-purpose Jelly in order to use it for Loom migrations, which may introduce complexities because of the advanced features of the work (i.e., Jelly provides more functionalities such as call graph construction; not only pattern matching) than developing a new tool for code transformation.

CogniCrypt [19] detects misuses of cryptographic APIs using a DSL called CrySL. CrySL has a pre-defined set of rules for several libraries such as Java Cryptographic Architecture (JCA). CrySL also facilitates users to define their rule files, which are then used to detect violations of those code patterns. CogniCrypt is useful for detecting refactoring patterns in Java code and can be used for platform thread usage detection.

Hotfixer [22] is a tool developed to hotfix vulnerabilities found using CogniCrypt. It uses a patch adapter that depends on a custom-built version of Soot [41]. The code fixes happen at the bytecode level. However, our goal is to apply Loom transformations at the source level instead.

Most of these code transformation tools do not align with our goals for Loom migrations, because of language restrictions or having support for limited use cases. Hence, we decided to explore the options available to develop a tool for our requirements. We found that Soot can only provide output in Java/Android bytecode, Jimple, and Jasmine [41]. Our goal is to migrate application servers and evaluate performance differences after migration. Therefore, we need to perform transformations at the source level rather than obtaining the output of the migration in the byte code formats will make it difficult to generate source code.

Chapter 4

Migrating Java Programs to Loom

In this chapter, we present Loomizer and how it performs migrations using extracted refactoring patterns.

4.1 Refactoring Patterns

At first, we tried using existing work to mine refactoring patterns for Loom APIs, but we faced several obstacles because Loom is a new language feature. Hence, we resorted to manually extracting a set of refactoring patterns from the JDK enhancement proposal for virtual threads [7]. Table 4.1 depicts the patterns that we extracted and their corresponding transformations.

To create platform threads, developers use the public thread constructor with a few arguments as illustrated in the first four patterns in Table 4.1. Loom developers exposed a set of APIs for virtual threads instead of a public constructor. Developers should not pool virtual threads because of their lightweight and inexpensive nature that comes with comparatively less overhead in creating many more virtual threads. As the last two patterns show, developers may use `Executors` to assign a virtual thread per task using a `ThreadFactory` that creates virtual threads.

As we have explained in Chapter 3, prior work about automatic code transformation is not applicable to migration to Loom. To better understand how to build an automatic migration tool for Loom, we first explored manually

Table 4.1: Patterns and their corresponding transformations

Pattern	Transformation
`new Thread(runnable);`	`Thread.ofVirtual().unstarted(runnable);`
`new Thread(runnable, string);`	`Thread.ofVirtual().name(string).unstarted(runnable);`
`new Thread(threadGroup, runnable);`	`Thread.ofVirtual().unstarted(runnable);`
`new Thread(threadGroup, runnable, string);`	`Thread.ofVirtual().name(string).unstarted(runnable);`
`Executors.newFixedThreadPool(integer);`	`ThreadFactory threadFac = Thread.ofVirtual().factory();` `Executors.newThreadPerTaskExecutor(threadFac);`
`Executors.newCachedThreadPool();`	`ThreadFactory threadFac = Thread.ofVirtual().factory();` `Executors.newThreadPerTaskExecutor( threadFac );`

performing code transformations on Open Liberty using our extracted refactoring patterns. However, this was a tedious task, confirming the need for an automated tool chain to support migrating existing codebases to Loom.

4.2 Loomizer

We developed Loomizer, a tool that automatically migrates platform threads to virtual threads in Java software applications. Loomizer is based on the Rascal meta-programming language. We chose Rascal because it provides Java source-to-source transformation [34], [36] and pattern matching [29]. We also found that several existing works [2] have utilized Rascal in different use cases. For example, TestAXE [30] is an open-source tool for refactoring code smells in Java test code. Therefore, we extended TestAXE to develop Loomizer by building three main components: Driver, Main Program, and Code Transformer.

4.2.1 Driver

The Driver is the starting point of Loomizer and it accepts the root directory path to a Java source code, and recursively accesses each subfolder using a `glob()` Python module [32]. It then executes our Rascal main program using the Rascal standalone jar file [33]. To fix the format of the generated code, Driver uses the Google Java code formatter [13].

```
11 CompilationUnit unit;
12 try{
13   unit = parse(#CompilationUnit, code);
14 }
```

Figure 4.1: Parsing the Java code to CompilationUnit

```
15 list[Transformation] transformations = [
16 transformation("Loomizer", loomTransform)];
17
18 CompilationUnit transformedUnit;
19 for(loc f <- allFiles) {
20   try {
21     str content = readFile(f);
22     file = f;
23     <transformedUnit> = applyTransformations(content, transformations);
24     if (unparse(transformedUnit) != "") {
25       writeFile(f, transformedUnit);
26     }
27   }
28   catch: {
29     continue;
30   }
```

Figure 4.2: Applying transformations to Java files

4.2.2 Main Program

The main program filters out all input Java files and reads the content of each file using the `readFile()` function in Rascal. Next, the main program parses the file content to a parse tree that starts with a root node called `CompilationUnit`, as shown in Figure 4.1. `CompilationUnit` is the root syntax definition of a parse tree generated for any Java file, and it contains several children syntax definitions (i.e., children nodes) representing each element in a Java source code including keywords, variables, and literals. Rascal allows traversing through each node using `switch` and `visit` expressions. After applying Loom transformations on a `CompilationUnit` using the code transformer, the main program writes the transformed source code back to the Java file using the `writeFile()` function in Rascal as illustrated in Figure 4.2.

```
31 start syntax CompilationUnit = PackageDeclaration? Imports TypeDeclaration
       *;
```

Figure 4.3: The syntax definition of `CompilationUnit` in RASCAL.

```
32 syntax MethodDeclaration = methodDeclaration: MethodModifier* MethodHeader
       MethodBody ;
```

Figure 4.4: The syntax definition of `MethodDeclaration` in RASCAL.

4.2.3 Code Transformer

The code transformer replaces the detected patterns with the corresponding replacements by modifying the syntax definitions in the parse tree of the input Java program.

Parse Tree Representation

RASCAL enables pattern matching and code transformation through a parse tree [34] that represents each element in a Java source code. `CompilationUnit`, the root node of a parse tree generated for any Java code, contains three children nodes: `PackageDeclaration`, `Imports` and `TypeDeclaration` as shown in Figure 4.3. `PackageDeclaration` represents the package name that a Java file may contain. `Imports` represents all `import` statements in the input file. Each Java file may be either a Java class or an interface and its corresponding representation in the parse tree is `TypeDeclaration`.

`MethodDeclaration` is the corresponding RASCAL parse tree syntax representation for a Java method and it contains three children representations as shown in Figure 4.4. Figure 4.5 depicts the construction of each child node representing associated Java elements. Therefore, for the sample Java method in Figure 4.6, we identify the following facts:

- `public` → `MethodModifier`
- `void hello()` → `MethodHeader`
- {`System.out.println("Hello World!  ");`} → `MethodBody`

```
33 syntax MethodModifier = Annotation
34                       | "public"
35                       | "protected"
36                       | "private"
37                       | "abstract"
38                       | "static"
39                       | "final"
40                       | "synchronized"
41                       | "native"
42                       | "strictfp"
43                       ;
44
45 syntax MethodHeader = methodHeader: Result MethodDeclarator Throws?
46                     |  TypeParameters Annotation* Result MethodDeclarator
                           Throws?
47                     ;
48
49 syntax MethodBody = Block ";"*
50                   | ";"
51                   ;
```

Figure 4.5: The syntax definitions of children nodes of `MethodDeclaration` in Rascal.

```
52 public void hello() {
53     System.out.println("Hello World! ");
54 }
```

Figure 4.6: The Java source code of a sample input method.

```
55 case (MethodInvocation) `<ExpressionName exp>.getId()` => (MethodInvocation
     ) `<ExpressionName exp>.threadId()`
```

Figure 4.7: Replacing code in Rascal.

```
56 case (MethodInvocation) `Thread.currentThread().getId()` : { insert((
     MethodInvocation) `Thread.currentThread().threadId()`); }
```

Figure 4.8: Inserting code with code execution block in Rascal.

Using the parse tree representation for Java and pattern matching techniques in Rascal including `visit` and `switch` statements, we detect mined refactoring patterns by traversing through tree nodes.

Java Source-to-Source Transformation

Rascal facilitates two ways for code transformations either using `=>` or `:`. As Figure 4.7 shows, `=>` allows replacing a statement with another statement only if both of them are of the same type. Using `:` enables executing any code block on the righthand side, whether it may or may not be a replacement. The `insert` expression within a code block can replace the statement on the lefthand side only if the replacement has the same type, as shown in Figure 4.8.

We identified that there are several situations where the patterns and their corresponding transformations do not have matching data types to perform code transformations directly with either available operator. For example, to create a platform thread using a public thread constructor, `new Thread(runnable)` is of type `ClassInstanceCreationExpression`, while `Thread.ofVirtual().unstarted(runnable)` is a `MethodInvocation`. To choose a common parent node for both cases to perform the replacement, we need to consider three types of statements: `BlockStatement`, `StatementExpression`, and `ReturnStatement`. In Loomizer, as demonstrated in Figure 4.9, we have separate `case` expressions for each of the statement types so that we can detect the patterns and perform transformations at the parent node level.

For `new Thread(<ArgumentList args>)`, we need to identify the data types of the arguments passed into the constructor to perform the relevant transfor-

```
case (BlockStatement) `Thread <VariableDeclaratorId id> = new Thread(<
    ArgumentList args>);` : {
    BlockStatement blockstatementExp = (BlockStatement) `Thread <
        VariableDeclaratorId id> = new Thread(<ArgumentList args>);`;
    println("blockstatementExp: <blockstatementExp>");
}
case (ReturnStatement) `return new Thread(<ArgumentList args>);` : {
    ReturnStatement returnSte = (ReturnStatement) `return new Thread(<
        ArgumentList args>);`;
    println("returnStatement : <returnSte>");
}
case (StatementExpression) `<LeftHandSide id> = new Thread(<ArgumentList
    args>)` : {
    StatementExpression exp = (StatementExpression) `<LeftHandSide id> = new
         Thread(<ArgumentList args>)`;
    println("statementExpr : <exp>");
}
```

Figure 4.9: Thread creation API in multiple syntax definitions

```
public class HelloClass {
    String name = "Anne"; //"name" is a class variable
    hello(name); //"name" is an argument

    public void hello(String name) { //"name" is a parameter (method
        parameter)
  String prefix = "Hello World! "; // "prefix" is a local variable
        System.out.println(prefix + name);
    }
}
```

Figure 4.10: Data types of variables in a Java code

mations. The arguments are references to variables, and parameters declared in the Java source code itself and we cannot directly determine the data types using the Rascal parse tree. Therefore, we have developed an approach to identify the data types of arguments.

Identifying Data Types of Arguments

There are 3 types of variables [27] in a Java source code: class variables, method parameters, and local variables as shown in Figure 4.10. The references to these variables stand as arguments in the public thread constructor.

- Class variables: are declared inside a class, outside methods, and blocks. All objects in a class can access class variables. In LOOMIZER, we con-

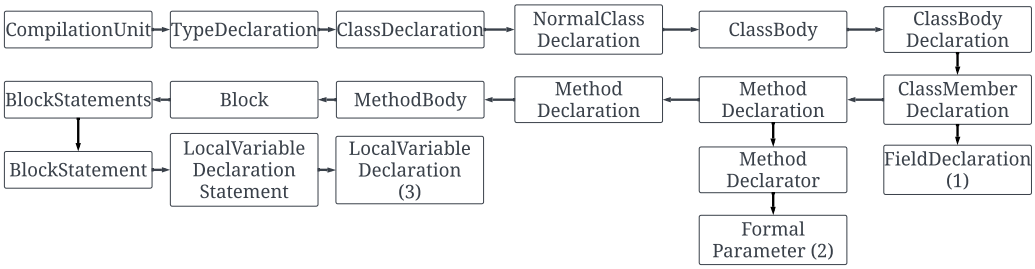

Figure 4.11: Traversal through nodes to extract data types of variables or parameters.

```
78 syntax FieldDeclaration = fieldDeclaration: FieldModifier* UnannType
       VariableDeclaratorList ";"+ ;
79
80 syntax FieldModifier = Annotation
81                      | "public"
82                      | "protected"
83                      | "private"
84                      | "static"
85                      | "final"
86                      | "transient"
87                      | "volatile"
88                      ;
89
90 syntax VariableDeclaratorList = variableDeclaratorList: {VariableDeclarator
        ","}+ ;
91
92 syntax VariableDeclarator = variableDeclarator: VariableDeclaratorId ("="
       VariableInitializer)? ;
```

Figure 4.12: The syntax definition of `FieldDeclaration` and its children nodes in Rascal.

```
94 case FieldDeclaration f: {
95   UnannType vType;
96   VariableDeclaratorId name;
97   f = top-down visit(f) {
98     case UnannType s: {
99       vType = s;
100    }
101    case VariableDeclaratorId s: {
102      name = s;
103    }
104  }
105  classVariableNameTypeMap += (name : vType);
106 }
```

Figure 4.13: Extracting class variables by visiting `FieldDeclaration`.

```
case FormalParameter f : {
  UnannType vType;
  VariableDeclaratorId name;
  f = top-down visit(f) {
    case UnannType s: {
      vType = s;
    }
    case VariableDeclaratorId s: {
      name = s;
    }
  }
  variableNameTypeMap += (name : vType);
}
```

Figure 4.14: Extracting parameters by visiting FormalParameter

```
syntax FormalParameter = VariableModifier* mds UnannType atype
    VariableDeclaratorId vdid;
syntax VariableModifier = Annotation
                        | "final"
                        ;
```

Figure 4.15: Syntax Definition of FormalParameter and its children nodes

```
case LocalVariableDeclaration lvd: {
  UnannType vType;
  VariableDeclaratorId name;
  lvd = top-down visit(lvd) {
    case UnannType s: {
      vType = s;
    }
    case VariableDeclaratorId s: {
      name = s;
    }
  }
  variableNameTypeMap += (name : vType);
}
```

Figure 4.16: Extracting parameters by visiting `LocalVariableDeclaration`.

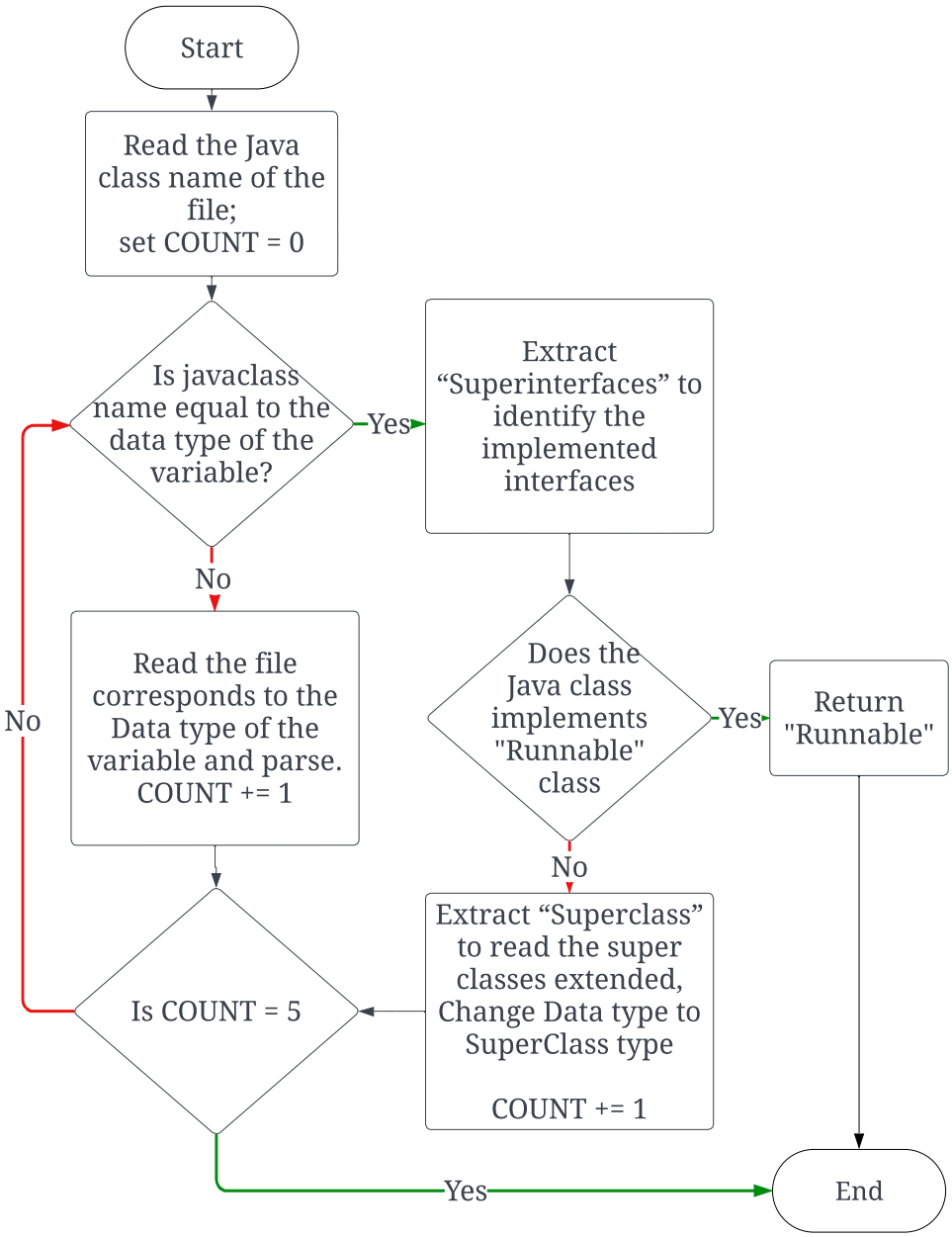

Figure 4.17: The main algorithm in LOOMIZER that figures out the class hierarchy relationship between a given class and `Runnable`.

sider both instance variables and static variables under class variables. `FieldDeclaration` is the corresponding syntax that represents class variables in RASCAL. Figure 4.11 shows traversal through nodes using the arrow keys starting from `CompilationUnit` to `FieldDeclaration` which is labeled as (1). Figure 4.12 depicts the construct of `FieldDeclaration` using multiple children nodes and Figure 4.13 shows how the code transformer extracts the name and the data type of the variables and stores them in a local map.

- Method Parameters: are declared in the method signature as depicted in Figure 4.10. `MethodDeclaration` in a parse tree represents a method, and `FormalParameter` represents its parameters. `FormalParameter` comes within `MethodDeclarator` and Figure 4.11 depicts the traversal path to `FormalParameter` (2) . We extract the data type and the name of the parameter and store them in a separate local map as shown in Figure 4.14, because `FormalParameter` contains a similar construct as `FieldDeclaration` according to the RASCAL syntax definitions shown in Figure 4.15.

- Local Variables: are declared inside a method. According to the parse tree generated with RASCAL, `LocalVariableDeclaration` (3) represents local variables, as shown in Figure 4.11. We can identify the data type and the name of the local variables as demonstrated in Figure 4.16.

We maintain local maps inside LOOMIZER so that we can refer to it whenever we need to find the data type of a particular argument. There are more complex scenarios where the data type of variables is not as obvious (e.g., `Thread`, `String`, `Runnable`, `ThreadGroup`), which means it is a custom class type defined in the application itself. This custom class may implement a Java interface such as `Runnable` or extend another class which may finally implement the `Runnable` runnable. Hence, we consider the data type as `Runnable` for our transformations. To correctly find types of those custom data types, we have implemented an algorithm to verify whether they are eventually of

```
138 list[ArgumentList] argumentList = [];
139 top-down visit(statement) {
140   case ArgumentList argList : argumentList += argList;
141 }
```

Figure 4.18: Extracting arguments from the statements

type `Runnable`, as shown in Figure 4.17. Following our examination of the source codes of the application servers, we made a few assumptions to develop this algorithm. Our algorithm assumes that the parent class files (extended classes) will be present in the same package itself. It also assumes that the data type can be found within five iterations. If the algorithm cannot find a relationship with `Runnable` even after five iterations, it ignores that type and relevant transformation. It is not a common scenario we have encountered in our source codes.

Detection and Transformation of Refactoring Patterns

Patterns **P1**, **P2**, **P3**, and **P4** use the public thread constructor to create platform threads where the only difference is the number of arguments and data types of the arguments. These patterns can be found in three types of statements as depicted in Figure 4.9. Regardless of the statement type, LOOMIZER extracts the argument list passed to the constructor as shown in Figure 4.18. LOOMIZER then identifies the data type of each argument using the following strategy:

- Find the argument name in the local maps created earlier for different types of variables.
- Remove the `this` keyword from arguments, before finding them in local maps (e.g., `this` in `this.variableName`). In that scenario, we remove the prefix and only extract the argument name.
- Arguments may contain built-in methods such as `x.toString()`. In this case, the type of the argument `x` is `String`.
- Arguments may be integer, boolean, or string (i.e., primitive data types).

```
142 //statement expression
143 StatementExpression replacingExpression = (StatementExpression) `<
        LeftHandSide id> = Thread.ofVirtual().unstarted(<ArgumentList lambdas>)
        `;
144
145 //return statement
146 ReturnStatement replacingExpression = (ReturnStatement) `return Thread.
        ofVirtual().unstarted(<ArgumentList lambdas>);`;
147
148 //block statement
149 BlockStatement replacingExpression = (BlockStatement) `Thread <
        VariableDeclaratorId id> = Thread.ofVirtual().unstarted(<ArgumentList
        lambdas>);`;
150
151 insert(replacingExpression);
152 }
```

Figure 4.19: Performing transformation of **P1**.

- Arguments may be method calls. We maintain a separate local map in LOOMIZER to store the method names and their return types. Searching through the map enables us to find the data type of such arguments.
- Arguments may contain a concatenation of multiple variable references. LOOMIZER splits them and identifies the type of any.

Once LOOMIZER has identified the data type of all arguments to method calls of interest, it applies the following pattern transformations:

Pattern P1: `new Thread(runnable)` If the type of the argument is `Runnable`, LOOMIZER generates the replacement expressions according to the statement type as demonstrated in Figure 4.19. To generate a replacement expression, LOOMIZER constructs the required argument list. Because of the processing required to generate the replacement expression, LOOMIZER does not use `=>`. It instead uses the RASCAL operator `:` with an `insert` expression.

Pattern P2: `new Thread(runnable, string)` In this pattern, there are two arguments: one is a `Runnable` and the other one is a `String`. If LOOMIZER finds a match for this pattern, it arranges the arguments as needed for the replacement expression. In the replacement expression, we have to parse and use

```
153 //statement expression
154 StatementExpression replacingExpression = (StatementExpression) `<
        LeftHandSide id> = Thread.ofVirtual().name(<ArgumentList nameArgs>).
        unstarted(<ArgumentList runnableArgs>)`;
155
156 //return statement
157 ReturnStatement replacingExpression = (ReturnStatement) `return Thread.
        ofVirtual().name(<ArgumentList nameArgs>).unstarted(<ArgumentList
        runnableArgs>);`;
158
159 //block statement
160 BlockStatement replacingExpression = (BlockStatement) `Thread <
        VariableDeclaratorId id> = Thread.ofVirtual().name(<ArgumentList
        nameArgs>).unstarted(<ArgumentList runnableArgs>);`;
161
162 insert(replacingExpression);
```

Figure 4.20: Performing transformation of **P2**.

two `ArgumentList` objects for thread name and the `runnable` object separately as illustrated in Figure 4.20, and generate distinct replacement expressions depending on the statement type. We also use the `insert` function to apply the generated replacement expression.

Pattern P3: `new Thread(threadGroup, runnable)` Thread groups are not active with virtual threads [7]. Therefore, this pattern is similar to **P1**, because the only argument that LOOMIZER needs to consider is `runnable`.

Pattern P4: `new Thread(threadGroup, runnable, string)` In this pattern, LOOMIZER does not consider the thread group argument. Therefore, it is also similar to **P2**.

Pattern P5: `Executors.newFixedThreadPool(integer)` To create thread pools with a fixed number of threads, LOOMIZER uses Java `Executors`. With virtual threads, we should not create thread pools because they are lightweight and inexpensive. With virtual threads, we can generate a thread per task. Therefore, the transformation creates an instance of `ThreadFactory`, which is capable of generating virtual threads. It then passes the `threadFactory` instance to the new API exposed via virtual threads, as shown in Figure 4.21.

```
163 case (MethodInvocation) `Executors.newFixedThreadPool(<ArgumentList args>)
      `: {
164   MethodInvocation methodInv = (MethodInvocation) `Executors.
        newFixedThreadPool(<ArgumentList args>)`;
165   list[ArgumentList] argumentList = [];
166   top-down visit(methodInv) {
167     case ArgumentList argList : argumentList += argList;
168   }
169   ...
170   str variableNameForThreadFac = "threadFactory";
171   ArgumentList threadFactoryArgs = parse(#ArgumentList,
        variableNameForThreadFac);
172   MethodInvocation replacingExpression = (MethodInvocation) `Executors.
        newThreadPerTaskExecutor( <ArgumentList threadFactoryArgs> )`;
173   insert(replacingExpression);
174 }
```

Figure 4.21: Performing transformation of **P5**.

We do not need multiple instances of `ThreadFactory` within a single method. Hence, LOOMIZER inserts the instance creation at the beginning of the method. Figure 4.22 depicts the simple parsing implementation that we developed in LOOMIZER to accomplish the insertion of thread factory instance creation. We unparse the method body, remove the curly braces that wrap the method body, and insert the expression at the top.

Pattern P6: `Executors.Executors.newCachedThreadPool()` Similar to **P5**, we use the instance of `ThreadFactory` to generate the replacement expression.

Patterns **P1**–**P6** represent the six patterns that LOOMIZER currently considers for migrating platform thread usages to virtual threads. LOOMIZER additionally transforms two more patterns that are not related to Loom, but are essential for the compatibility with Java 19. This is because Java maintainers deprecated the method for obtaining the identity of a thread and introduced the method `threadId()` instead in Java 19 [26]. Similarly, when accessing the identity of the current thread, we have to use the method `threadId()` instead of `getId()` as shown in **P8**.

Pattern P7: `thread.getId()` To transform this pattern, LOOMIZER uses its local maps to identify the data type of `thread` extracting only the variable

```
175 VariableDeclaratorId vId = parse(#VariableDeclaratorId,
        variableNameForThreadFac);
176
177 BlockStatement statementToBeAdded = (BlockStatement) `ThreadFactory <
        VariableDeclaratorId vId> = Thread.ofVirtual().factory();`;
178
179 //unparse method body
180 str unparsedMethodBody = unparse(methodB);
181
182 // replace first and last curly braces with empty strings
183 unparsedMethodBody = replaceFirst(unparsedMethodBody, "{", "");
184
185 ...
186
187 // construct the method body again
188 str methodBody = "{\n" + unparse(statementToBeAdded) + "\n" +
189 unparsedMethodBody + "\n}";
190
191 MethodBody newBody = parse(#MethodBody, methodBody);
192
193 // insert new method body
194 insert(newBody);
```

Figure 4.22: Instantiation of a `ThreadFactory` instance.

name as depicted in Figure 4.23.

Pattern P8: `Thread.currentThread().getId()` Figure 4.24 shows the corresponding `case` statement and transformation for this pattern.

We have recreated the code examples in this chapter for the comprehensibility of the readers. Therefore, they may not contain the exact code that appears in the source code of Loomizer. Because these transformations include code insertions and replacements that affect the format of the code, we have also integrated `google-code-format` [13] into Loomizer, similar to TestAXE [30]. In addition to applying Loomizer on Java source code, when working with Java 19 source code, we had to enable preview features because virtual threads are a preview feature in Java 19.

```
case (MethodInvocation) `<ExpressionName exp>.getId()` : {
  MethodInvocation mi = (MethodInvocation) `<ExpressionName exp>.getId()`;
  MethodInvocation mi2 = (MethodInvocation) `<ExpressionName exp>.threadId
      ()`;
  println("methodInvocation : <mi> detected");
  bool threadIdUseFound = false;
  top-down visit(mi) {
    case ExpressionName exp: {
      for(VariableDeclaratorId vId <- variableNameTypeMap) {
        str unparsedExp = trim(unparse(exp));
        if (startsWith(unparsedExp, "this.")) {
          unparsedExp = substring(unparsedExp, 5);
        }
        if (trim(unparse(vId)) == unparsedExp) {
          if (trim(unparse( variableNameTypeMap[vId] )) == "Thread") {
            threadIdUseFound = true;
            break;
          }
        }
      }
    }
  }
  if (threadIdUseFound) {
    datetime transformedTime = now();
    println("methodInvocation : <mi2> transformed : <transformedTime>");
    insert((MethodInvocation) `<ExpressionName exp>.threadId()`);
  }
}
```

Figure 4.23: Detection and Transformation of **P7**.

```
case (MethodInvocation) `Thread.currentThread().getId()` : {
  println("methodInvocation : detected");
  insert((MethodInvocation) `Thread.currentThread().threadId()`);
}
```

Figure 4.24: Detection and Transformation of **P8**.

Chapter 5

Evaluation

In this chapter, we present an empirical evaluation to observe the benefit of migrating to Loom for several application servers in two main scenarios: serving IO-bound applications and serving CPU-consuming applications.

5.1 Experimental Setup

5.1.1 Application Servers

To execute and host software applications, developers use various application servers as the runtime environment. In this thesis, we automatically migrate platform thread usages to virtual threads for the following Java-based application servers: IBM Open Liberty [24], Apache Tomcat [39], Wildfly (formerly known as JBoss) [42], and Undertow [40].

Using virtual threads may improve application throughput. Therefore, we want to observe throughput values before and after migrations. For that, we have built two versions of each application server: one with the original source code cloned from the master/main branch (e.g., `integration` branch in Open Liberty) in its repository and the other with the migrated source code resulting from Loomizer.

5.1.2 WRK Benchmarking Tool

To evaluate throughput changes before and after migration, we utilize the WRK load driver [43]. The driver generates a large number of requests on multiple cores. We can fine-tune the number of threads, connections, and duration

for a test to generate the required load of requests. For example, the command `wrk -t25 -c100 -d30s http://127.0.0.1:8080/index.html` runs WRK with a total number of 25 threads (`-t25`), and a total number of 100 HTTP open connections (`-c100`), for 30 seconds(`-d30s`). At the end of test execution, WRK produces an output that contains the total number of requests that reached the endpoint and the total number of requests processed per second (i.e., throughput).

5.1.3 Machine Setup

We ran all performance tests on a Ubuntu 22.04.3 LTS server with x86-64 architecture and 64 CPU cores. Depending on the objective of each experiment, we used CPU affinity to limit the number of CPU cores assigned. Due to the heavy load generated by WRK, the CPU on which WRK runs, comes to immediate saturation. Therefore, to configure the WRK load driver for heavy loads, we have to monitor the application status, CPU, and memory consumption. To achieve that, we utilized Nigel's performance Monitor for Linux (NMON) data collector and visualizer [14] which records all CPU, memory, and disk-related metrics with all the processes-related information.

To observe any prolonged pause times caused by stop-the-world events, we employed IBM's Garbage Collection (GC) and Memory Visualizer [15] to analyze GC logs collected throughout all the experiments because stop-the-world pauses the entire execution and may impact performance. We restricted external interference to our performance tests by terminating all the running Java, WRK, and NMON processes before each test execution. Furthermore, we limited all other variables that might change for different tests (e.g., define a custom heap size instead of using default values). We then analyzed the collected GC logs for each experiment to adjust heap sizes based on the recommendations given by the IBM GC and Memory Visualizer.

To maintain the stability of the data, we had 20 test runs per experiment and calculated the median value using the output of all test runs. We modified a publicly available Python parser [35] to generate a CSV file using the WRK output files so that we can use the CSV file to compute the median values.

We have also applied a warm-up load for 5 minutes before the real load test which runs for 3 minutes to allow the Just-In-Time (JIT) compiler to process any hot code paths.

For the repeatability of the tests, we wrote separate shell scripts for each application server. Each shell script contains a set of tasks: killing Java and other processes, starting up the server, executing the WRK command, writing the output to a file, and terminating the server. We have made Loomizer, including our refactoring patterns, experiment scripts, and data available [37].

Once we have all the data including median throughput before and after migrations, we compute the percentage of throughput improvement as:

$$\text{Throughput Improvement} = \left(\frac{\text{TA} - \text{TB}}{\text{TB}}\right) \times 100\%$$

Where:

$$\text{TA} : \text{Throughput After Migration}$$

$$\text{TB} : \text{Throughput Before Migration}$$

5.2 Research Questions

According to the JDK Enhancement Proposal [7], virtual threads are beneficial in several scenarios. It also suggests that we can observe application throughput improvement in the following two conditions (hereafter, referred to as JEPrules):

- the number of concurrent tasks is high
- the workload is not CPU-bound

To understand the throughput differences of Loom migrated application servers, we conducted an empirical evaluation by answering the following research questions constructed based on JEPrules.

RQ1: How efficient is Loomizer?

RQ2: Does Loom migration improve throughput for applications with mostly IO-bound operations?

RQ3: Does Loom migration improve throughput for applications with a mix of both IO-bound and CPU-bound operations?

5.3 Deployed Applications

We developed a Spring Boot Java application [4] with endpoints deployable on each application server to accept requests generated from the WRK load driver. We included several blocking operations in the application to evaluate how virtual threads improve application throughput during wait times. Our application contains a sleep operation within a `for` loop that runs 5 times acting as an IO-bound scenario. Later during our experiments, we have modified this delay to observe the throughput improvement with varying delays.

Each application server requires the deployed application to have different dependencies and plugins so they can be successfully deployed. Hence, we created clones of the application, added those dependencies and plugins accordingly, and built their WAR files separately.

To answer RQ2 and RQ3, we will use the following types of applications:

Scenario 1 (S1): Application with mostly IO-bound tasks Under this scenario, our deployed application is not CPU-bound. According to JEPrules, we might need more concurrent tasks to benefit from using virtual threads. However, we cannot identify the number of concurrent tasks processed using WRK. Hence, we consider throughput to be more than a few thousand (i.e., at least, about 2,500 or more requests per second) as an alternative proxy for serving a large number of concurrent tasks. To obtain a higher value for throughput, we need to adjust the number of threads/connections in WRK configurations. To achieve that, we used 50 ms as the lowest delay in the sleep operation in the application.

Scenario 2 (S2): Application with a mix of both IO-bound and CPU-bound operations According to JDK enhancement proposal [7], the workload should not be CPU-bound and the presence of IO blocking operations

Table 5.1: Code migration patterns and their occurrences in the source code of the application servers

Patterns	Open Liberty	Tomcat	Wildfly	Undertow
`new Thread(Runnable)`	23	19	2	3
`new Thread(Runnable, String)`	19	3	1	0
`Executors.newCachedThreadPool()`	3	0	1	0
`Executors.newFixedThreadPool(int)`	9	1	19	11
`new Thread(ThreadGroup, Runnable)`	1	0	0	0
`new Thread(ThreadGroup, Runnable, String)`	1	4	0	0
Total Migrations	**56**	**27**	**23**	**14**

in an application is mandatory to observe application throughput improvement with virtual threads because virtual threads are not faster than platform threads. Hence, we wanted to observe the impact of virtual threads on applications with both IO-bound and CPU-bound operations. Therefore, we modified our application by introducing two CPU-consuming algorithms: reversing a long text and counting the number of occurrences of a letter in a long text. With these modifications, our application becomes a mix of CPU-bound and IO-bound tasks, because the application contains both blocking operations and CPU-consuming operations. Since we might not observe an application throughput improvement when the system under test is merely CPU-bound, under this scenario, we maintain higher CPU utilization across all the application servers to observe the impact of virtual threads with blocking operations and CPU-consuming operations. To achieve that, we started our experiments with 0 ms, at which the CPU utilization of all running CPU cores is nearly 100%. When choosing the appropriate WRK configuration, we also managed to achieve higher CPU utilization at 25 ms. WRK did not have to produce a much higher load of incoming requests to increase CPU utilization.

5.4 How efficient is Loomizer? (RQ1)

In this section, we assess the advantage of using Loomizer over manual migration, which includes manually searching all usages of platform threads in a source code and converting them to support loom APIs. Figure 5.1 and Figure 5.2 illustrate two example transformations applied on Open Liberty and

Figure 5.1: An example transformation applied on Open Liberty source code

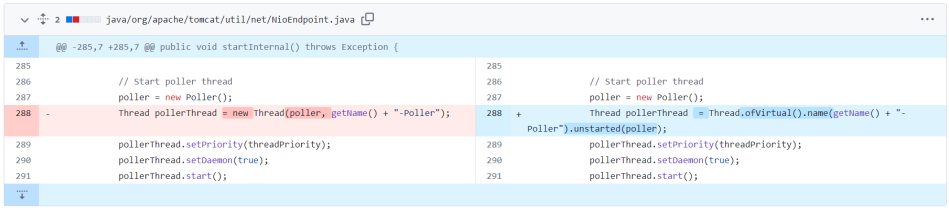

Figure 5.2: An example transformation applied on Tomcat source code

Tomcat using Loomizer. Table 5.1 depicts the number of Loom transformations completed using Loomizer on each application server with the number of performed transformations of each pattern in each server code base. Overall, Loomizer has applied 56 transformations on the Open Liberty source code and 27 on the Tomcat source code. Hence, if we used manual migration instead of Loomizer, we have to spend a lot of time locating instances (i.e., 8 patterns) in large code bases. Moreover, if we find another pattern, we do not have to go through each LOC in the code base again to locate usages, instead, we can define the pattern in LOC to automate the process.

Table 5.2 depicts the number of LOC in each original application server and time Loomizer spent on each server during migration. We noticed that the time taken to finish the migrations of each application server varies greatly. Loomizer has taken approximately 26 hours to complete automatic migration of Open Liberty, but around an hour for Undertow. We compared the time spent by Loomizer on migrations of each application server and the number of LOC in the source code of each application server before migration. According to data in Table 5.2, when the number of LOC is higher, Loomizer spends more time to complete. Loomizer is based on Rascal and it parses each Java file to a parse tree. `CompilationUnit` is the root node in a parsed Java file

Table 5.2: Number of LOC in Application Server vs Time spent on Migrations

Application Server	LOC	Time (hours)
Open Liberty	11,670,792	25.9
Tomcat	238,691	2.60
Wildfly	1,354,663	4.30
Undertow	234,867	0.91

according to the Java grammar [34] in Rascal and we need to start from it to traverse through every other element. Parsing each and every file in a Java application that contains many modules and a larger number of LOC, and traversal through each node to accomplish pattern matching, will be time-consuming and that should be the reason for Loomizer spending more hours with Open Liberty and comparatively fewer hours with Undertow.

We have also manually validated the accuracy of the transformations applied using Loomizer on each application server. From that, we observed that, Loomizer has transformed all the transformations correctly. Moreover, Loomizer does not report any false positives; no any irrelevant detection other than platform thread usages.

> Loomizer is efficient over manual migration because it automatically detects and transforms platform threads to virtual threads using a set of refactoring patterns.

5.5 Does Loom migration improve throughput for applications with mostly IO-bound operations? (RQ2)

In this section, we describe the experimental setups used in the WRK load driver and with the application servers.

We start our experiments with 50 ms as the lowest delay and progressively increase it to 100 ms, 200 ms, 500 ms, 1,000 ms, 1,500 ms, 2,000 ms, 2,500 ms, and 3,000 ms. In these experiments, we do not consider less than 1% throughput improvement because of the variation in the sample values.

Table 5.3: Delay vs Throughput Before and After Migration in Open Liberty

Delay	Before Migration (requests/s)	After Migration (requests/s)
50 ms	4,898.88	5,369.84
100 ms	4,880.39	5,620.73
200 ms	4,525.66	5,307.71
500 ms	3,862.34	4,369.40
1,000 ms	2,919.46	3,150.22
1,500 ms	2,144.84	2,267.47
2,000 ms	1,648.66	1,717.53
2,500 ms	1,328.87	1,369.43
3,000 ms	1,103.73	1,127.75

As discussed earlier under S1, we have to maintain a higher throughput as our common objective. Hence, WRK requires a higher amount of threads and connections in its configuration to produce the target throughput. Due to the heavy number of threads and connections in WRK, it leads to immediate saturation of the CPU core on which WRK runs. Therefore, we increased the number of CPU cores assigned to WRK and also employed more than one instance of WRK to maximize the load of requests depending on the application server.

5.5.1 Open Liberty

Configuration

We deployed the application on Open Liberty by placing the generated war file inside the `apps` directory in the built Open Liberty. We assigned two CPU cores for the Open Liberty application server to run on because it should increase concurrent tasks. After fine-tuning with IBM GC and memory visualizer, we defined the maximum and minimum heap sizes as 2,560 MB and 256 MB to limit external interference. We also conducted the performance tests with all delays under the same configuration. We then generated a throughput of about 5,000 requests/s at 50 ms with a single instance of WRK and 2 CPU cores assigned.

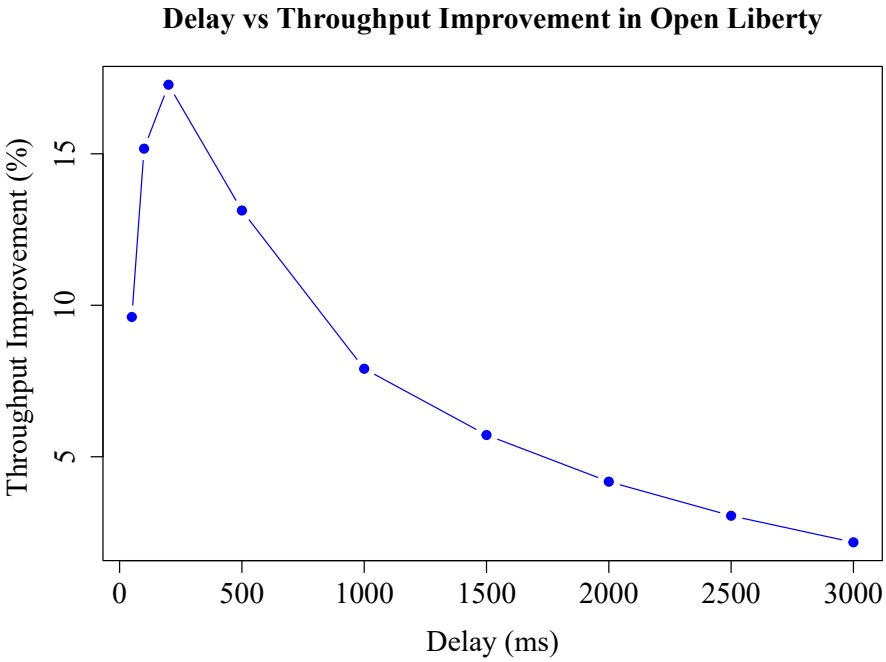

Figure 5.3: Delay vs Throughput Improvement in Migrated Open Liberty Server

Results

In Open Liberty, we observed that throughput decreases with increasing delay at a slightly constant rate according to values shown in Table 5.3. We calculated the throughput improvement using the equation described previously. Figure 5.3 shows that at 50 ms delay, throughput improvement is about 10% and keeps increasing with increasing delay till 200 ms. The maximum throughput improvement is approximately 17% at 200 ms delay in the blocking operation. More increments to delay do not further increase throughput improvement. Throughput improvement decreases with increasing delay after 500 ms. At all delays starting from 50 ms to 3,000 ms, we observed throughput improvements. At 3,000 ms, throughput improvement is nearly 2.18%. Overall, the throughput improvement at 3,000 ms is about one-fourth of the throughput improvement at 50 ms.

Discussion

Open Liberty produces much higher throughput compared to other application servers [5]. Since having a slightly constant rate of decrement over varying delays, throughput is in the thousands even at a 3,000 ms delay in the blocking operation and displays throughput improvement. Regardless of the delay, we only observed throughput improvements after migration, which means that the benefit of virtual threads is promising in the presence of blocking operations. After 200 ms, throughput improvement decreases because the overhead (e.g., having many threads in the blocked state) of virtual threads increases.

5.5.2 Tomcat

Configuration

We placed the generated WAR file for our deployed application in a folder called `webapps/output/build` under the build directory for Tomcat.

With the same WRK configuration used for Open Liberty, we could not observe a similar amount of throughput with Tomcat at 50 ms. For S1, since we have to maintain a higher throughput, we changed the WRK configuration for Tomcat by introducing more instances and adjusting the number of CPU cores allocated to the Tomcat server. When we have four WRK instances and each WRK instance contains four CPU cores assigned, we could observe a higher throughput value without saturating the CPU cores assigned to WRK under the heavy load of requests. Moreover, we had to employ three CPU cores on which the Tomcat server runs because a single CPU core could not handle that much of incoming requests.

For this experiment, we limited the maximum heap size to 1,080 MB after analyzing the collected GC logs, because the longer pause time may suspend the execution of other threads. This suspension may impact performance raising scalability concerns.

Table 5.4: Delay vs Throughput Before and After Migration in Tomcat

Delay	**Before Migration (requests/s)**	**After Migration (requests/s)**
50 ms	3,011.97	2,985.46
100 ms	1,359.44	1,389.59
200 ms	668.18	667.51
500 ms	244.90	246.39
1,000 ms	106.65	106.42
1,500 ms	59.74	60.44
2,000 ms	36.88	36.75
2,500 ms	22.32	22.51
3,000 ms	13.31	14.04

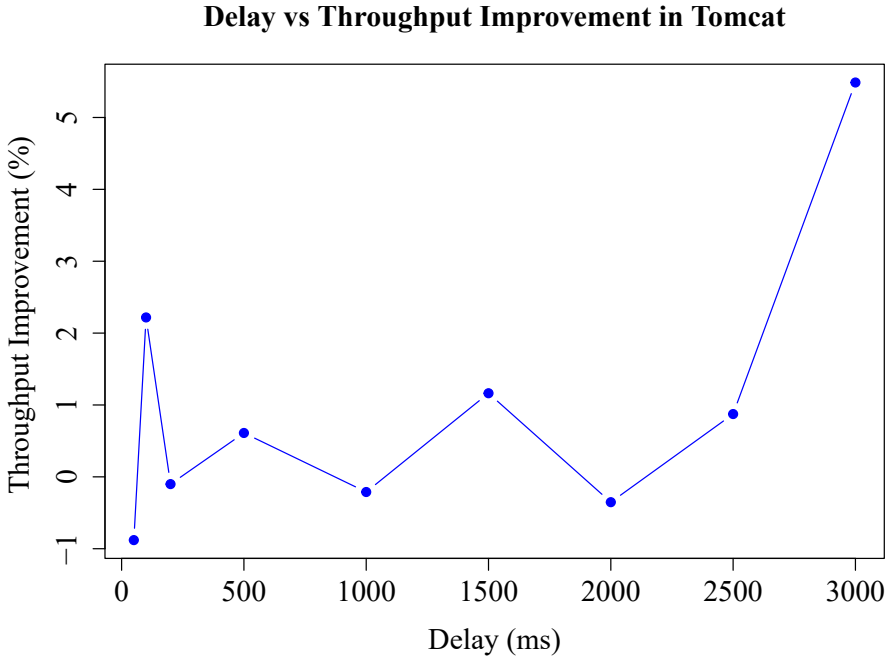

Figure 5.4: Delay vs Throughput Improvement in Migrated Tomcat Server

Results

Unlike Open Liberty, with the introduction of a 100 ms delay, throughput drops drastically to half of that of 50 ms in Tomcat as shown in Table 5.4. Throughput drop decreases with increasing delay. Figure 5.4 shows that at lower delays, we observed a fluctuation between -1% and 2%. At a delay of 3,000 ms, we observed a maximum throughput improvement of 5.4%.

Discussion

Observing a considerable throughput improvement at higher delays means that the benefits of virtual threads are promising. Throughput improvement was not prominent at lower delays because having many virtual threads might have introduced a considerable overhead that decreased the benefits of virtual threads.

5.5.3 Wildfly

Configuration

We placed the application war file into `standalone/deployments` in the built Wildfly directory to perform deployment on Wildfly.

Though we observed more than 2,500 requests/sec throughput with Open Liberty and Tomcat at 50 ms, we could not easily achieve the same goal for Wildfly. To achieve that, we had to employ 10 WRK instances where each instance has 3 CPU cores to produce our target throughput. We also assigned 16 CPU cores to the Wildfly server because the heavy load of requests from WRK could not be handled with fewer CPU cores. To minimize external interference, we set the minimum and maximum heap sizes as 256 MB and 1,080 MB, respectively.

Results

We observed that throughput at 50 ms is approximately 3,500 requests/sec and drops more than half at a 100 ms delay as depicted in Table 5.5. With the drop, we observed that we cannot increase the delay beyond 2,500 ms because

Table 5.5: Delay vs Throughput Before and After Migration in Wildfly

Delay	Before Migration (requests/s)	After Migration (requests/s)
50 ms	3,458.03	3,447.77
100 ms	1,473.74	1,468.64
200 ms	657.20	662.94
500 ms	206.71	205.06
1,000 ms	76.74	76.93
1,500 ms	34.47	35.17
2,000 ms	15.41	15.41
2,500 ms	4.26	4.27

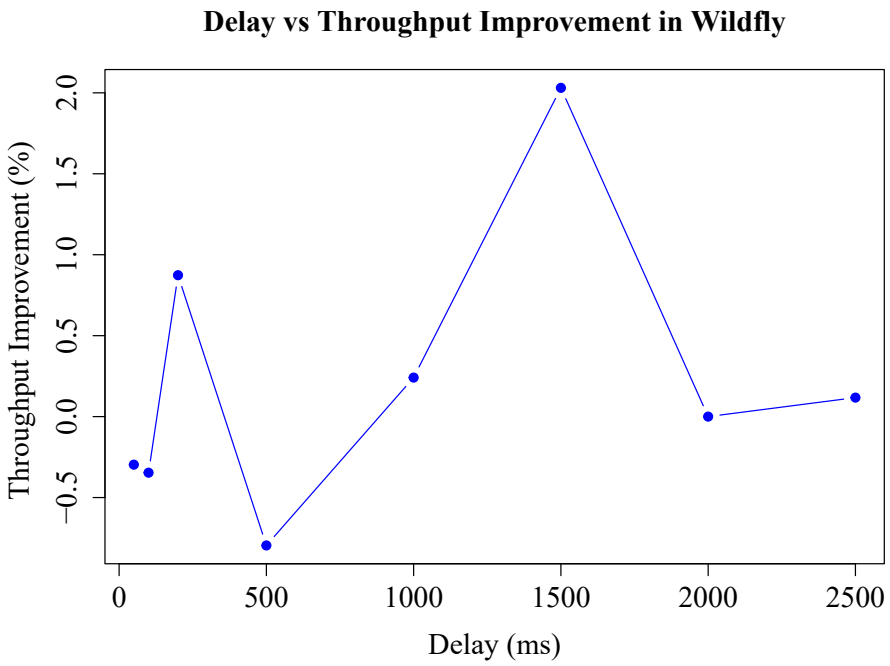

Figure 5.5: Delay vs Throughput Improvement in Migrated Wildfly Server

it produces zero throughput at a 3,000 ms delay. At lower delays, we observed a fluctuation in throughput improvement of less than ± 1%, which we can ignore due to variation in sample values, and the maximum throughput improvement of slightly above 2% at 1,500 ms as depicted in Figure 5.5. Later, there is a drop in throughput improvement with increasing delay to 2,000 ms and showing near-equal throughput values before and after migration at 2,500 ms.

Discussion

At higher delays, we observed throughput improvement, which means virtual threads may improve application throughput at higher delays in the blocking operations with Wildfly. Further increasing delay may introduce additional overhead from virtual threads and lead to diminishing returns depicting near-zero throughput improvement after 1,500 ms.

5.5.4 Undertow

Configuration

The Undertow server does not work as a standalone server, unlike other application servers. It should be embedded within the Java application. We built the Undertow server from its migrated and original source code and put the path to the built version in the `pom.xml` build file of the sample application. Similar to starting up the server before each test execution, with Undertow, we started the application using the `mvn` command to expose the endpoints.

Similar to the Wildfly server, to obtain our target of around 2,500 requests/sec throughput at 50 ms, we fine-tuned the setup. Eight WRK instances and four CPU cores in each instance could produce the target throughput. Since the application with an embedded Undertow server on a single CPU core could not handle many incoming requests, we employed 26 CPU cores for that. To limit external interference, we set maximum and minimum heap sizes as with other application servers.

Table 5.6: Delay vs Throughput Before and After Migration in Undertow

Delay	**Before Migration (requests/s)**	**After Migration (requests/s)**
50 ms	2,382.43	2,375.09
100 ms	1,097.68	1,097.59
200 ms	478.27	477.89
500 ms	133.59	133.78
1,000 ms	24.35	24.67

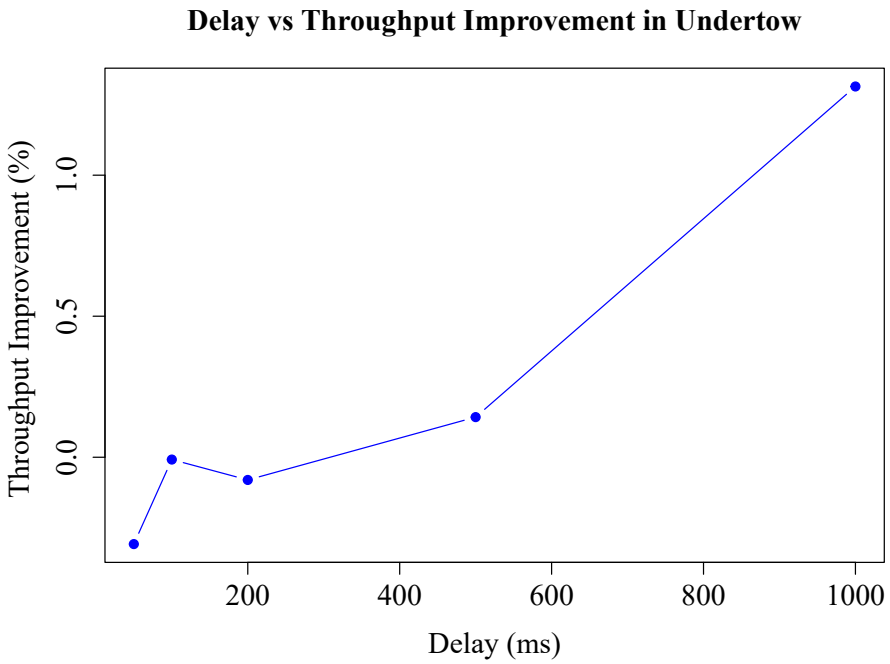

Figure 5.6: Delay vs Throughput Improvement in Migrated Undertow Server

Results

After fine-tuning the setup, we observed about 2,382 requests/sec at 50 ms. With increasing delay, throughput drops and reaches near 24 requests/sec at 1,000 ms as depicted in Table 5.6. Hence, we observed that throughput is zero for delays greater than 1,000 ms. Though we achieved the target throughput at 50 ms using a setup that consumes the highest number of cores, we could not perform experiments beyond 1,000 ms because of the throughput drops. As demonstrated in Figure 5.6, throughput improvement increases with increasing delay showing a throughput deterioration of 0.3% at 50 ms. At 1,000 ms, we observed the maximum throughput improvement which is about 1.3% compared to the original Undertow server.

Discussion

Though we employed many CPU cores for this experiment to observe the target throughput at 50 ms, due to the drop in throughput over the delay, we could not observe non-zero throughput after 1,000 ms. Throughput improvement with the Loom migrated Undertow server indicates that virtual threads may improve application throughput during higher wait times.

> Loom migration improves throughput for applications with mostly IO-bound operations by 1%–17%. The improvement depends on the application server (e.g., functions that use threading mechanisms) and the delay in the blocking operations because blocking operations are the key to improving throughput using virtual threads.

5.6 Does Loom migration improve throughput for applications with a mix of both IO-bound and CPU-bound operations? (RQ3)

According to JEPrules, we cannot observe throughput improvement with virtual threads when the workload is CPU-bound. Hence, under this research

question, we explore the throughput improvement, when we have both blocking and CPU-consuming operations in an application (i.e., S2). In this scenario, We maintain higher CPU utilization instead of higher throughput. Maintaining higher CPU utilization with CPU-consuming tasks is not very challenging, hence, we do not need a heavy load of requests as used with S1. Because of that, we employed only one instance of WRK with two CPU cores. We also assigned only one CPU core for the application server because it easily leads to higher CPU utilization rather than having many. At 0 ms (no delay) in the blocking operation, we could observe that CPU utilization of CPU cores, on which each application servers run, is close to 100%. Hence, in selecting WRK configurations, we also considered the CPU utilization at 25 ms.

Similar to the previous experiments, we ignore the throughput improvement of less than 1% considering the variation in the sample values.

5.6.1 Open Liberty

Configuration

We started our experiments setting 0 ms as the lowest delay at which we do not have blocking operations in the application. Hence, we assumed that the scenario would become CPU-bounded at 0 ms because the application consists of only CPU-consuming operations. Then, we incrementally changed the delay to 25 ms, 50 ms, 100 ms, 200 ms, 500 ms, 1,000 ms, 1,500 ms, 2,000 ms and 2,500 ms.

Results

Table 5.7 depicts throughput is in thousands for delays from 0 ms to 200 ms which implies there are higher concurrent tasks. As demonstrated in Figure 5.7, we observed about 20% throughput improvement at 0 ms, though we expected it would become CPU-bound at 0 ms and show throughput deterioration according to JEPrules. For 50 ms, 100 ms, and 200 ms delays in the blocking operations, we discovered throughput improvement of around 5-6%. After 1,500 ms, the throughput difference between the original and migrated source codes is approximately zero.

Table 5.7: Delay vs Throughput Before and After Migration in Open Liberty

Delay	Before Migration (requests/s)	After Migration (requests/s)
0 ms	1,349.92	1,619.36
25 ms	1,288.50	1,312.04
50 ms	1,183.20	1,258.98
100 ms	1,082.19	1,155.37
200 ms	949.86	1,015.45
500 ms	577.28	594.20
1,000 ms	306.19	298.05
1,500 ms	205.28	202.46
2,000 ms	154.12	152.15
2,500 ms	122.20	122.07

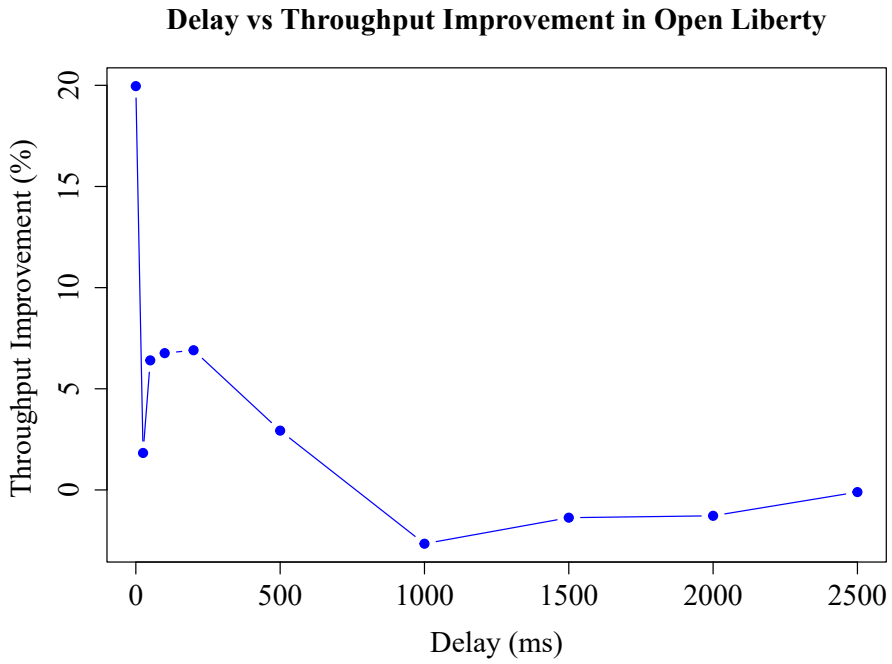

Figure 5.7: Delay vs Throughput Improvement in Migrated Open Liberty Server

Table 5.8: Delay vs Throughput Before and After Migration in Tomcat

Delay	Before Migration (requests/s)	After Migration (requests/s)
0 ms	1,404.82	1,159.77
25 ms	1,374.18	1,399.46
50 ms	795.90	796.00
100 ms	398.34	398.49
200 ms	198.99	199.09
500 ms	79.34	79.43
1,000 ms	39.70	39.68
1,500 ms	26.43	26.43
2,000 ms	19.62	19.76
2,500 ms	15.55	15.55

Discussion

Ideally, at 0 ms, the situation would become CPU-bound because we do not have waiting times set in our application. The reason for our observation of higher application throughput improvement at 0 ms is having blocking operations in the Open Liberty source code itself with higher concurrent tasks. It directs our scenario to become a mix of IO-bound and CPU-bound instead of solely CPU-bound at 0 ms. We only observed throughput improvements at lower delays till 200 ms due to blocking operations in the application and Open Liberty server resulting in higher wait times in requests. Unlike S1, we could not observe a longer range of delays where throughput improvement is present with S2.

5.6.2 Tomcat

Configuration

Choosing WRK configurations for this experimental setup was easier compared to S1 because we do not have to generate a heavy load with WRK to observe a higher CPU utilization. We could observe near 100% and 99% CPU consumption at 0 ms and 25 ms delays in the application.

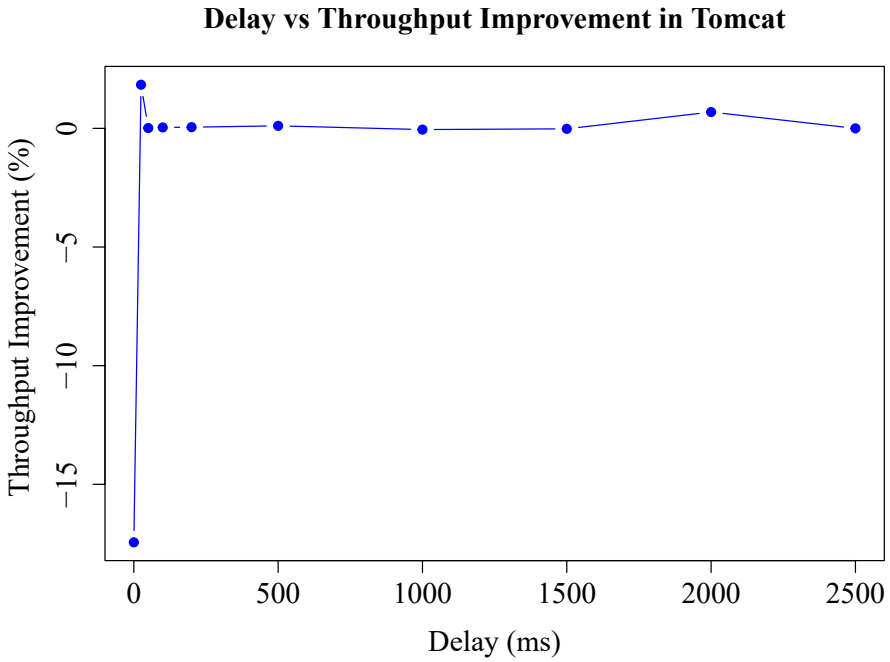

Figure 5.8: Delay vs Throughput Improvement in Migrated Tomcat Server

Results

At 0 ms, the throughput after Loom migrations is lower than before migration; throughput deterioration, as shown in Table 5.8. The system under test is CPU-bound at 0 ms because Tomcat and the deployed application do not contain blocking operations. It proves that throughput improvement with virtual threads is not prominent with merely CPU-bound applications. The maximum throughput improvement visible is also nearly close to 1.8%; less than 2%, which is at 25 ms as depicted in Figure 5.8. We observe a fluctuation around zero for all other delays displaying almost equal throughput before and after migration at 2,500 ms.

Discussion

At 0 ms, the CPU core is almost saturated, and virtual threads cannot improve application throughput because virtual threads are not faster than platform threads [7]. That is why we do not see any throughput improvement at 0 ms.

Throughput at 25 ms is in the thousands, hence, there is a higher number of

Table 5.9: Delay vs Throughput Before and After Migration in Wildfly

Delay	Before Migration (requests/s)	After Migration (requests/s)
0 ms	438.25	438.23
25 ms	120.53	120.54
50 ms	62.25	62.25
100 ms	31.41	31.41
200 ms	15.68	15.68
500 ms	6.17	6.17
1,000 ms	2.98	2.98

concurrent tasks which results in a throughput improvement at 25 ms. Though there are higher wait times in the application, it does not show any throughput improvement greater than 1% after 25 ms, probably due to the lack of enough concurrent tasks.

5.6.3 Wildfly

Configuration

Though we observed nearly 100% CPU utilization at 0 ms with Wildfly, the maximum CPU utilization at 25 ms was about 57%. Increasing the number of WRK instances or WRK configuration did not increase CPU utilization further at 25 ms. In this experiment also, we could not observe non-zero throughput for larger delays; after 1,000 ms, because of the huge throughput fall observed with the introduction of a delay.

Results

As depicted in Figure 5.9, there was a fluctuation between ± 0.02 % at lower delays. At 200 ms onwards, throughput improvement is almost zero; throughput before and after migration is equal.

Discussion

Our results suggest that we cannot see application throughput improvement with virtual threads deploying S2 application on the Wildfly server. With S2, we focused on maintaining higher CPU utilization than higher throughput in choosing WRK configurations. Table 5.9 depicts that throughput at 0 ms is

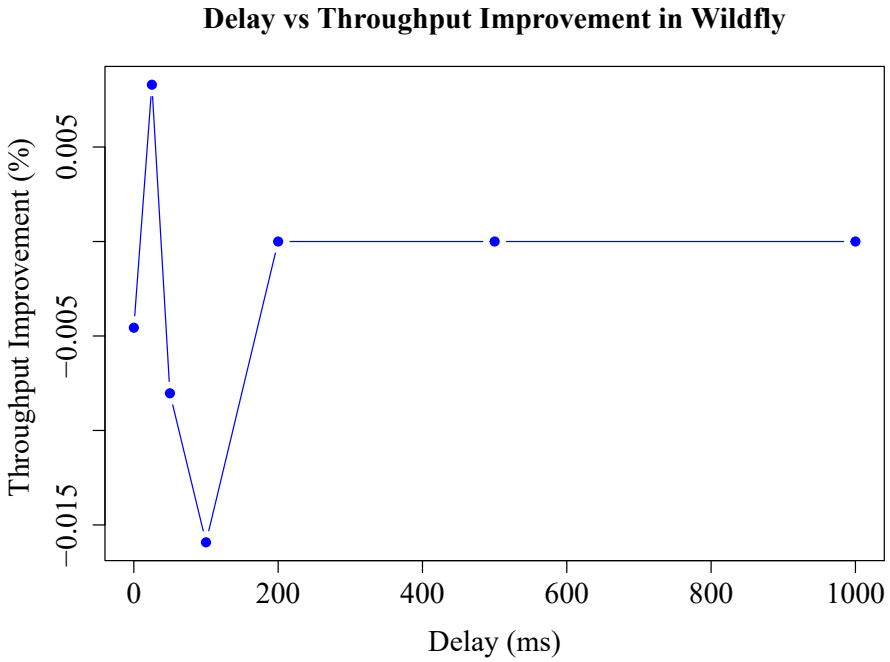

Figure 5.9: Delay vs Throughput Improvement in Migrated Wildfly Server

about 440 requests/s. Therefore, we might not have enough concurrent tasks required to observe the benefit of virtual threads though the CPU is not fully utilized. At 0 ms, we observe throughput deterioration because the system is CPU bounded and according to JEPrules, virtual threads cannot improve application throughput when the workload is CPU bounded.

5.6.4 Undertow

Configuration

Similar to Wildfly, we could observe almost 100% CPU utilization at 0 ms, however, at 25 ms, the maximum CPU utilization was 57%. We could not experiment with more than a 1,000 ms delay in the application because it resulted in zero throughput.

Results

We observed a throughput improvement of less than 0.4% at 0 ms, which is negligible because of the fluctuation of sample data. Till 200 ms, there is

Table 5.10: Delay vs Throughput Before and After Migration in Undertow

Delay	Before Migration (requests/s)	After Migration (requests/s)
0 ms	298.56	299.60
25 ms	119.44	119.45
50 ms	60.53	60.52
100 ms	31.11	31.05
200 ms	15.68	15.68
500 ms	6.17	6.17
1,000 ms	2.98	2.98

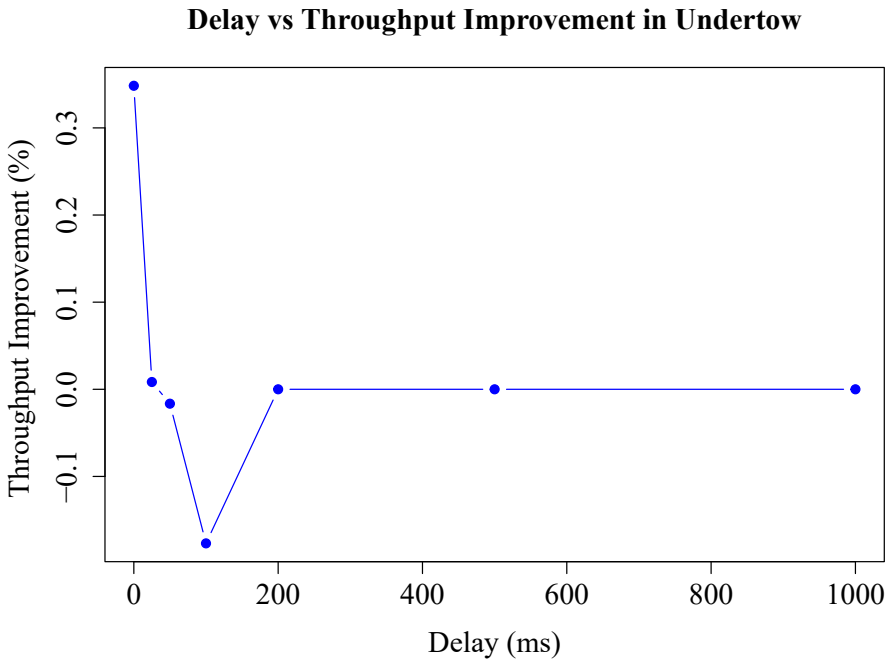

Figure 5.10: Delay vs Throughput Improvement in Migrated Undertow Server

fluctuation ± 0.3%, around zero as shown in Figure 5.10. After that, the throughput improvement remains stable at zero.

Discussion

At 0 ms, throughput was about 300 requests/s with the Undertow server as depicted in Table 5.10. Since our focus is on higher CPU utilization in S2, the incoming load of requests is also comparatively smaller; the number of concurrent tasks might also be low. Even though we increase the delay in the application, due to not having enough concurrent tasks and having limited hardware support resulting from busy CPUs, we cannot observe the benefit of virtual threads in terms of throughput.

> The S2 application with non-zero delay in blocking operations does not benefit much from virtual threads except with Open Liberty. The presence of blocking operations in the Open Liberty source code itself and in the application would be the reason for displaying a higher throughput improvement at delays starting from 0 ms to 200 ms with Open Liberty.

5.7 Discussion

As mentioned in the JEPrules that we discussed earlier, there are two conditions that make us observe the benefits of virtual threads by improving application throughput. When we use an application that contains merely sleep operations, out of those two conditions, we have to satisfy one; the number of concurrent tasks should be more than a few thousand because the CPU is not bounded when the application is merely sleeping. Hence, under S1, we decided to maintain a higher throughput (around 2,500 requests/sec at 50 ms) for all the experiments conducted with each application server using the S1 application. We observed that throughput decreases with increasing delay. Throughput improvement increases with increasing delay until the maximum throughput improvement in each application server with fluctuations as shown

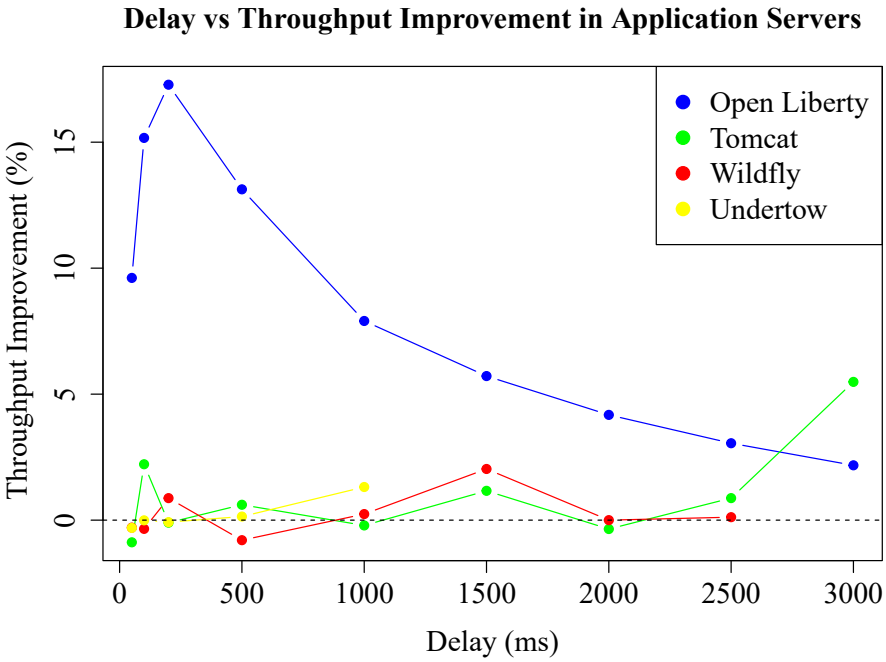

Figure 5.11: Delay vs Throughput Improvement in all Servers

in Figure 5.11.

We also observed that having a higher throughput does not always improve the application throughput with virtual threads. The overhead of virtual threads is comparatively low because virtual threads are lightweight. When we consider many number of virtual threads, there might be a considerable overhead of virtual threads. Hence, Different factors including the potential overhead of virtual threads, and the wait time (i.e., delay in the blocking operation) set in the blocking operation decide throughput improvement/deterioration compared to platform threads, though the number of concurrent tasks is high and the CPU is not bounded.

With each application server, we noticed a maximum throughput improvement at a certain delay. The delays at which a particular application server shows throughput improvement changes from server to server. Hence, the application throughput improvement relies on the application server as well. Additionally, with increasing delay, throughput improvement decreases because of diminishing returns, as a result of having a large number of virtual

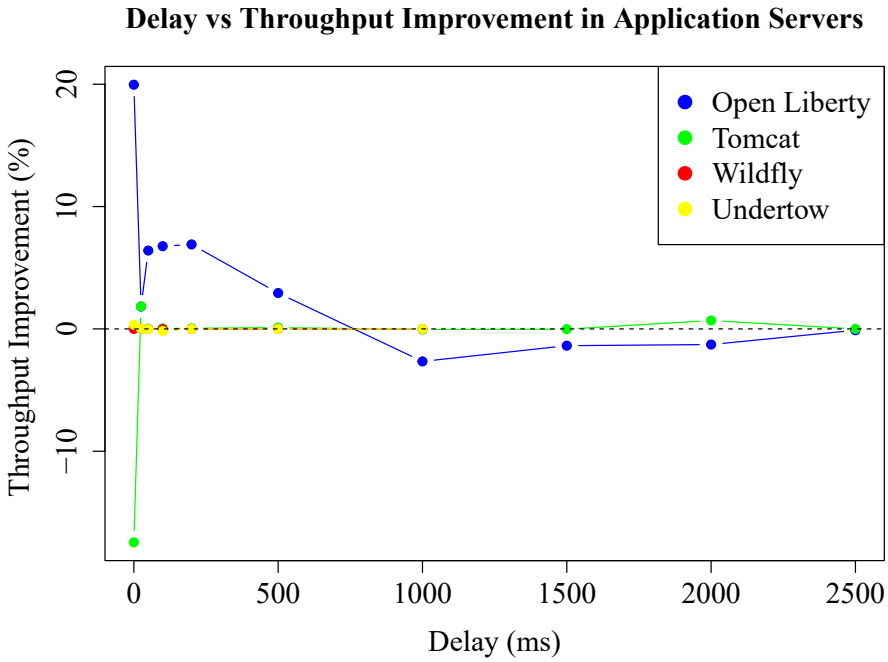

Figure 5.12: Delay vs Throughput Improvement in all Servers (with the application which contains both IO-bound and CPU-bound tasks)

threads in the blocked state with higher delays.

When we experimented with the S2 application, we observed that except for Open Liberty, any other application server did not show much throughput improvement at 0 ms. When there is no delay in the application and we deploy the application on Open Liberty and Tomcat, both report throughput in the thousands. At 0 ms, we observed a throughput improvement only with Open Liberty because of the blocking operations (sleep operations) in the source code of Open Liberty. Tomcat, Wildfly, and Undertow application servers do not have blocking operations in their source code itself. Hence, at 0 ms, the system under test with the S2 application deployed on Open Liberty is not solely CPU bound; instead a mix of both CPU bound and IO bound. Other application servers are still CPU-bound at 0 ms because they do not have blocking operations, which is why they do not report throughput improvement with virtual threads at 0 ms.

Even in the presence of blocking operations, due to having busy CPUs and

a lower number of concurrent tasks, throughput improvement in the migrated application servers except in Open Liberty and Tomcat was not noticeable. Considering all these facts and observations, we found various factors (e.g., the type of application (whether merely sleeping or CPU-bound), the application server, the expected real-time load of requests, and the delay in the application) that decide the ability of virtual threads to improve application throughput.

Overall, though we satisfied JEPrules with virtual threads, we could not always observe throughput improvement with S1. Hence, developers of a particular software application should decide whether they need to migrate their application to support virtual threads or not analyzing the consequences on the performance of their application.

5.8 Limitations

Loomizer does not currently support several complex scenarios that may require handling carefully with available Rascal syntax for Java, for example, the arguments can be lambda expressions with a code block execution or anonymous implementation of `Runnable` interface. In those cases, the common scenario of having a code block becomes the barrier that makes it harder to encode and transform. Additionally, Loomizer does not work for cases such as if a variable refers to a custom class defined outside the package because it requires retrieving the content from the particular file, which may need to manipulate the file path using `imports` statements.

5.9 Threats to Validity

We manually extracted refactoring patterns from JDK enhancement proposal [7]. Hence, there might be patterns that we may have missed. Since we performed an empirical evaluation to provide an overview of the effects of Loom migrations using a set of patterns, the conclusions might not drastically change. In addition, we migrated the Undertow application server to support Java 19 and to Loom, we assume that it may not have broken any functionalities.

Chapter 6

Conclusion

With the evolving nature of language features, language maintainers introduce new language features from time to time to make the language feasible for consumers in developing their software applications. Virtual threads are one of the language features introduced with Java 19. Manual migration; manually extracting refactoring patterns, then, identifying their occurrences and transforming; might not be an easy task. There are several studies focused on automatic migration with different use cases and languages. Any of those work could not be used to mine refactoring patterns because loom is a new feature and we cannot find sufficient commit usages in open-source repositories. Hence, we presented a set of migration patterns that we manually extracted using JDK enhancement proposal [7]. In detecting and transforming according to the migration patterns, we found several research work relatable. However, none of the tools proposed in those were capable of satisfying our requirements due to limitations in languages and use cases. Therefore, in this thesis, we presented Loomizer, a tool that is capable of automatically detecting and transforming traditional thread usages to loom APIs. We migrated several application servers such as Open Liberty, Tomcat, Wildfly, and Undertow using Loomizer. We also measured and analyzed the time spent by Loomizer on each application server migration. We observed that Loomizer spends more time on Open Liberty and Open Liberty is the largest code base out of those four servers in terms of the number of lines of code. Loomizer does the parsing of each Java file to a rascal syntax element and due to Open Liberty

having a larger number of lines of code, it may have taken more time for Open Liberty.

Developers may be concerned about the changes in performance after migration because performance is one of the most important factors that decide the success of the application [6]. We evaluate the performance changes in the migrated application servers by conducting several experiments. We developed two spring boot applications to deploy on each application server; one has solely blocking operations and the other one has CPU-consuming operations along with blocking operations. The first application simulates an IO-bound application while the second one is a mix of both CPU-bound and IO-bound operations. We employed the WRK load driver to generate a load of requests. We considered two scenarios to conduct our empirical evaluation; one with the IO-bound application maintaining higher (more than a few thousand) throughput, and the other one with the other application that contains both blocking and CPU-consuming operations maintaining higher CPU utilization. We performed our experiments for several delay points in the blocking operations. In the first experimental scenario, we observed throughput improvements after migration in several delays in all application servers. The point at which maximum throughput improvement happens is different from server to server. Overall, we observed throughput improvement with the merely sleeping application (S1) deployed on every application server. However, in the other scenario, when we do not have a delay in the blocking operation, the application only contains CPU-consuming tasks, which of course makes it merely CPU-bound. In Open Liberty, we observed that there was a throughput improvement at that point of delay (no delay in the application) and then, we figured out that there were blocking operations in the server source code itself, which may have affected the throughput improvement. That means, in Open Liberty, when we do not have a delay in the application, the system under test is not solely CPU-bound as we assumed, because of the blocking operations in the server. With S2, though there are blocking operations, we could not observe a considerable throughput improvement in any application server except with Open Liberty. The combination of both blocking operations in

the Open Liberty server and the application may result in more benefit from virtual threads than any other servers.

In conclusion, only having virtual threads and satisfying JEPrules does not improve application throughput, instead, the application server, the type of the application, the load generated with the load driver, and the delay in the blocking operation are some factors that decide the ability to improve the application throughput according to our observations. Therefore, we cannot conclude whether the virtual threads are beneficial in each software application and developers should assess the performance of their systems considering the properties and requirements of their applications.

Moreover, we observed that although Loomizer migrated the API usages properly, there is still more involved work to change the thread model in the underlying codebase because the main intent of virtual threads is to employ thread-per-request model. This more involved change is beyond the scope of traditional, automated API migration tools. While we could measure throughput improvement a couple of cases, the current implementation of Loomizer shows that traditional API migration is not sufficient to reap the benefits of using virtual threads in application servers.

Further, researchers can extend Loomizer to support many language features as a future advancement while addressing any limitations found in Loomizer and introducing any other patterns/corner cases that may have been missed in developing Loomizer.

References

[1] *@babel/parser*, https://babeljs.io/docs/babel-parser, Accessed: May. 07, 2024.

[2] *Bibliography*, https://www.rascal-mpl.org/docs/Bibliography/, Accessed: May. 07, 2024.

[3] E. Blech, A. Grishchenko, I. Kniazkov, et al., "Patternika: A pattern-mining-based tool for automatic library migration," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2021, pp. 333–338. DOI: 10.1109/ISSREW53611.2021.00098.

[4] *Building an application with spring boot*, https://spring.io/guides/gs/spring-boot, Accessed: May. 15, 2024.

[5] G. Charters and T. Pickett, *How does open liberty's performance compare to other cloud-native java runtimes*, https://openliberty.io/blog/2022/10/17/memory-footprint-throughput-update.html, Accessed: May. 10, 2024, 2022.

[6] L. Collaborative Consulting, *Consequences of poorly performing software systems*, https://www.immagic.com/eLibrary/ARCHIVES/GENERAL/COLLABUS/C140206C.pdf, Accessed: May. 10, 2024, 2014.

[7] 2. O. Corporation, *Jep 425: Virtual threads (preview)*, https://openjdk.org/jeps/425, Accessed: May. 08, 2024, 2023.

[8] 2. O. Corporation, *Jep 428: Structured concurrency (incubator)*, https://openjdk.org/jeps/428, Accessed: May. 08, 2024, 2023.

[9] O. Corporation, *Jdk releases*, https://www.java.com/releases, Accessed: May. 10, 2024.

[10] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for api evolution," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 599–602. DOI: 10.1109/ICSE.2009.5070565.

[11] GitHub, *Top 50 programming languages globally*, Accessed: May. 23, 2024, 2024. [Online]. Available: https://innovationgraph.github.com/global-metrics/programming-languages.

[12] B. Goetz and B. Singh, *Introduction to java threads*, https://developer.ibm.com/tutorials/j-threads/, Accessed: May. 01, 2024, 2002.

[13] *Google-java-format*, https://github.com/google/google-java-format/blob/master/scripts/google-java-format-diff.py, Accessed: May. 09, 2024.

[14] IBM, *Original nmon web page*, https://www.ibm.com/support/pages/original-nmon-web-page, Accessed: May. 10, 2024, 2023.

[15] *Ibm monitoring and diagnostic tools - garbage collection and memory visualizer*, Accessed: May. 02, 2024. [Online]. Available: https://www.ibm.com/docs/en/mon-diag-tools?topic=monitoring-diagnostic-tools-garbage-collection-memory-visualizer.

[16] J. Juneau, *Generics: How they work and why they are important*, https://www.oracle.com/technical-resources/articles/java/juneau-generics.html, Accessed: May. 23, 2024, 2014.

[17] A. Ketkar, O. Smirnov, N. Tsantalis, D. Dig, and T. Bryksin, "Inferring and applying type changes," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1206–1218. DOI: 10.1145/3510003.3510115.

[18] P. Klint, T. van der Storm, and J. J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE Computer Society, 2009, pp. 168–177. DOI: http://doi.ieeecomputersociety.org/10.1109/SCAM.2009.28.

[19] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2382–2400, 2021. DOI: 10.1109/TSE.2019.2948910.

[20] A. Møller, B. B. Nielsen, and M. T. Torp, "Detecting locations in javascript programs affected by breaking library changes," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. DOI: 10.1145/3428255. [Online]. Available: https://doi.org/10.1145/3428255.

[21] A. Møller, O. H. Veileborg, B. B. Nielsen, M. Miller, and W. Xu, *Jelly*, https://github.com/cs-au-dk/jelly, Accessed: May. 10, 2024, 2023.

[22] K. Newbury, K. Ali, and A. Craik, "Hotfixing misuses of crypto apis in java programs," in *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '21, Toronto, Canada: IBM Corp., 2021, pp. 73–82.

[23] B. B. Nielsen, M. T. Torp, and A. Møller, "Semantic patches for adaptation of javascript programs to evolving libraries," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 74–85. DOI: 10.1109/ICSE43902.2021.00020.

[24] *Open-liberty*, https://github.com/OpenLiberty/open-liberty, Accessed: Apr. 30, 2024.

[25] Oracle, *Annotation type functionalinterface*, https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html, Accessed: May. 23, 2024.

[26] Oracle, *Thread*, Accessed: May. 27, 2024. [Online]. Available: https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Thread.html%5C#threadId().

[27] Oracle, *Variables*, Accessed: May. 27, 2024. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html.

[28] Oracle Corporation, *Jdk 19 release notes*, https://www.oracle.com/java/technologies/javase/19-relnote-issues.html, Accessed: May. 10, 2024.

[29] *Pattern matching*, https://www.rascal-mpl.org/docs/Recipes/BasicProgramming/PatternMatching/, Accessed: May. 23, 2024.

[30] E. Paula and R. Bonifácio, "Testaxe: Automatically refactoring test smells using junit 5 features," Oct. 2022, pp. 89–98. DOI: 10.5753/cbsoft_estendido.2022.227655.

[31] R. Pressler, *State of loom*, https://cr.openjdk.org/~rpressler/loom/loom/sol1_part1.html, Accessed: May. 10, 2024, 2020.

[32] Python, *Glob — unix style pathname pattern expansion*, Accessed: May. 27, 2024. [Online]. Available: https://docs.python.org/3/library/glob.html.

[33] Rascal, *Download and installation*, Accessed: May. 27, 2024. [Online]. Available: https://www.rascal-mpl.org/docs/GettingStarted/DownloadAndInstallation/.

[34] Rascal, *Module lang::java::syntax::java18*, https://www.rascal-mpl.org/docs/Library/lang/java/syntax/Java18#lang-java-syntax-Java18-CompilationUnit, Accessed: May. 10, 2024.

[35] M. Smeets, *Http benchmarking using wrk. parsing output to csv or json using python*, https://technology.amis.nl/software-development/performance-and-tuning/http-benchmarking-using-wrk-parsing-output-to-csv-or-json-using-python/, Accessed: Apr. 30, 2024.

[36] *Source to source transformations*, https://www.rascal-mpl.org/docs/WhyRascal/UseCases/SourceToSource, Accessed: May. 23, 2024.

[37] themaplelab, *Loomizer*, https://github.com/themaplelab/Loomizer, Accessed: July. 11, 2024.

[38] F. Thung, H. J. Kang, L. Jiang, and D. Lo, "Towards generating transformation rules without examples for android api replacement," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 213–217. DOI: 10.1109/ICSME.2019.00032.

[39] *Tomcat*, https://github.com/apache/tomcat, Accessed: Apr. 30, 2024.

[40] *Undertow*, https://github.com/undertow-io/undertow, Accessed: Apr. 30, 2024.

[41] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99, Mississauga, Ontario, Canada: IBM Press, 1999, p. 13.

[42] *Wildfly*, https://github.com/wildfly/wildfly, Accessed: Apr. 30, 2024.

[43] *Wrk - a http benchmarking tool*, https://github.com/wg/wrk, Accessed: Apr. 30, 2024.

[44] Y. Xi, L. Shen, Y. Gui, and W. Zhao, "Migrating deprecated api to documented replacement: Patterns and tool," in *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, ser. Internetware '19, Fukuoka, Japan: Association for Computing Machinery, 2019, ISBN: 9781450377010. DOI: 10.1145/3361242.3361246. [Online]. Available: https://doi.org/10.1145/3361242.3361246.