*future internet*

MDPI

*Article*

# An Automatic Transformer from Sequential to Parallel Java Code

**Alessandro Midolo and Emiliano Tramontana ***

Dipartimento di Matematica e Informatica, University of Catania, 95125 Catania, Italy; alessandro.midolo@phd.unict.it
* Correspondence: tramontana@dmi.unict.it; Tel.: +39-095-7383008

**Abstract:** Sequential programs can benefit from parallel execution to improve their performance. When developing a parallel application, several techniques are employed to achieve the desired behavior: identifying parts that can run in parallel, synchronizing access to shared data, tuning performance, etc. Admittedly, manually transforming a sequential application to make it parallel can be tedious due to the large number of lines of code to inspect, the possibility of errors arising from inaccurate data dependence analysis leading to unpredictable behavior, and inefficiencies when the workload between parallel threads is unbalanced. This paper proposes an automatic approach that analyzes Java source code to identify method calls that are suitable for parallel execution and transforms them so that they run in another thread. The approach is based on data dependence and control dependence analyses to determine the execution flow and data accessed. Based on the proposed method, a tool has been developed to enhance applications by incorporating parallelism, i.e., transforming suitable method calls to execute on parallel threads, and synchronizing data access where needed. The developed tool has been extensively tested to verify the accuracy of its analysis in finding parallel execution opportunities, the correctness of the source code alterations, and the resultant performance gain.

**Keywords:** software architecture; design; code generation; parallelism; refactoring

check for updates

**Citation:** Midolo, A.; Tramontana, E. An Automatic Transformer from Sequential to Parallel Java Code. *Future Internet* **2023**, *15*, 306. https://doi.org/10.3390/fi15090306

Academic Editor: Xu Wang

Received: 10 August 2023
Revised: 24 August 2023
Accepted: 2 September 2023
Published: 8 September 2023

**Copyright:** © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

## 1. Introduction

The new generation of processors boasts several cores, which multi-threaded programs use. Parallelism has been used in different domains to improve efficiency and performance, e.g., malware detection [1], deep learning [2], vehicle navigation [3], healthcare systems [4], and other scenarios [5]. On the one hand, concurrent programming can speed up performance when working with huge amounts of data, on the other hand, developers face many challenges, such as thread safety, program correctness, and performance tuning [6,7]. Despite the numerous advantages of multi-thread programming, many developers tend to avoid using concurrency due to the intricacies involved in reasoning with data dependencies [8]. To preserve program accuracy, multiple threads have to be properly synchronized to avoid uncontrolled concurrent access to shared data. Moreover, the overhead stemming from starting and operating several threads should be checked to ensure performance gain [9].

The automatic transformation of sequential programs to perform operations in a parallel fashion is still an open issue since it requires an in-depth data dependence analysis, which could be significantly complex. Moreover, the transformation into a parallel version should be guided by some strategies that could likely bring performance gains once the parallel version executes.

The state of the art shows some notable approaches that can automatically change applications into parallel versions [10]. Some approaches only focus on recursive algorithms, which represent a narrow category of algorithms [11,12]. In [13], a tool was proposed to

refactor Array into ParallelArray; and in [14], an approach was proposed to check whether a stream pipeline could be run in parallel. Such approaches are only suitable for arrays and streams, respectively, while our proposal provides a wider spectrum of parallel execution opportunities. Previous approaches focused on only identifying some categories of statements [15,16], or refrained from introducing parallelism [17–22]; our analysis is more comprehensive. Moreover, most of the existing approaches use concurrent libraries that predate Java 8, whereas our work takes advantage of the latest Java libraries for concurrent computations, providing the additional benefit of integrating high-level API calls into the resulting source code, and ensuring minimal overhead when initiating new threads. Our fine-grained transformation only changes a small amount of the initial source code, providing developers with an easy-to-read new version.

Although libraries, both for Java and C++, are made available to developers for supporting parallel programming [23,24], it is up to the developers to determine which code snippets are prone to parallelism and how APIs can be used, whereas our approach automatically generates a parallel version of the source code. Several other approaches perform automatic parallel transformation in the executable code [25–27]. Although the performance gain is superb, the parallel version is hidden from the developer and cannot further modify it, e.g., to solve further requirements, improve some parts, etc.; conversely, our proposed approach generates the changed source code, making the transformations transparent and the code prone to any further changes.

We propose an automated approach that statically analyzes source codes to look for fragments of code that can be safely run in parallel to provide performance gain. The approach and corresponding tool use control flow analysis, data dependence analysis, and a control flow graph to identify execution paths that can be safely run in parallel (safety is checked according to Bernstein's conditions [28]), showing that such paths require considerable computational efforts before synchronization is needed, so that a performance gain is expected. Firstly, code is analyzed to find sequences of statements that could run in parallel, while adhering to some syntactic constraints (e.g., blocks of code in the same conditional branch, etc.). Secondly, data dependence analysis and control dependence analysis are carried out to check data dependencies among statements that could run in parallel. Two statements have a data flow dependence when the output set (i.e., the set of variables written) of the first statement contains data that are in the input set (i.e., the set of variables read) of the second statement [28–30].

Thirdly, a control flow graph (CFG) [31] is built, which represents all the paths that could be executed by a program. Such a CFG is then used to determine the two paths that might execute in parallel and to evaluate whether the statement numbers in each path are sufficiently large to possibly improve execution time. Finally, once a path that has passed all previous analysis steps has been found, the source code is refactored to run a new thread and insert synchronization points where needed. The code of the new version is generated automatically for developers to further modify or simply run it.

Our primary original contributions are as follows. We present a comprehensive approach that (i) automatically transforms sequential Java source code into parallel code (in previous literature, the authors mainly tackled executable code); (ii) undertakes a detailed static analysis of the source code, comprising control flow, data dependence analysis, evaluation of Bernstein's conditions, and the assessment of the computational effort required; (iii) employs an a priori evaluation of the potential benefits of new threads by computing execution paths; and (iv) evaluates the benefits performed according to the context of the instructions that are potential candidates for parallel execution and the estimated computational effort required by such instructions. We performed several experiments to assess the benefits and correctness of our approach. We report on an experiment where the analysis steps were automatically executed on an application; the results of several tests executed on the refactored parallel version show that our transformation preserves correctness.

The rest of the paper is organized as follows.

Section 2 shows the state of the art and a comparison with our approach. Section 3 introduces the general approach and a high-level algorithm for analyzing code. Section 4 shows the static code analysis, revealing the context of method calls and data dependence. Section 5 describes the construction of the CFG for the analyzed method and the evaluation of instructions for parallel execution. Section 6 presents the analysis and transformation performed on a sample application and the execution results. Section 7 comments on the results and the limitations of the approach. Finally, our conclusions are drawn in Section 8.

## 2. Related Works

Parallel computing is becoming more popular due to the growth of multi-thread hardware. Many papers have proposed automated tools designed to efficiently refactor sequential code into its parallel version.

In [11,12], the authors presented two approaches to apply Atomic refactoring and Collection refactoring [32] to refactor synchronized statements. They proposed the following transformations: converting Int to AtomicInteger, Long to AtomicLong, HashMap to ConcurrentHashMap, WeakHashMap to ConcurrentWeakHashMap, and HashSet to ConcurrentHashSet. The authors focused on modernizing existing parallel code by using the new libraries provided since Java 5. They carried out an effectiveness test to check the correctness of their modifications. In addition, Dig et al. [12] proposed a refactoring approach changing sequential recursive algorithms into a parallel version using ForkJoinTask; they assessed popular recursive algorithms to show the benefits in execution time. Moreover, in [33], the authors presented a tool to substitute the Thread class with the Executor class to allow the use of a thread pool at runtime.

Conversely, our approach proposes several innovative aspects: (i) the application of a concrete refactoring opportunity from sequential to parallel, injecting new threads into the execution; in comparison, the transformations discussed by the aforementioned approaches just update some class types; (ii) the transformation shown by our approach is definitely less invasive than the one proposed with the ForkJoinTask; indeed, our approach requires the update of the instruction that calls a method with the use of a CompletableFuture and the addition of synchronization statements with the join() call; (iii) the applicability of our approach accepts all method calls that meet the shown preconditions, while the aforementioned approaches are relevant for recursive algorithms, primitive variables, some collections, and Java synchronized blocks.

Another refactoring approach was presented in [18], where the authors proposed a Lock refactoring approach to automatically refactor built-in monitor locks for Java's synchronized blocks, with the locks provided by the java.util.concurrent.locks library: ReentrantLock and ReadWriteLock types. An analysis was performed to check whether the transformations preserved the behavior of the application and if the updated locks guaranteed a performance boost. A similar approach was presented in [20,33], where a tool was developed to automatically transform synchronized locks to re-entrant locks. In [20], the authors presented an automated approach to convert a synchronized statement lock into a StampedLock. In [21], the authors proposed a prototype to automatically convert a coarse-grained lock into a fine-grained lock to reduce lock contention and, hence, improve performance and scalability. These approaches focus on modernizing and optimizing existing parallel code; hence, the developer has to decide which part of the code should be run in parallel. Differently, our approach takes a sequential code as input and finds and introduces parallel constructs to boost performance and scalability.

A practical eclipse-based tool was presented in [17], which replaced the global mutable state with a thread-local state, and introduced a thread to run the refactored code in parallel. The aim of the tool is to reduce the number of executions that share the same input and, hence, increase the parallelization opportunities. This approach involves invasive changes in the source code since global states are removed/moved, and the boilerplate code is inserted to create a thread. Conversely, our approach fits the code analyzed, inserting the synchronization in the proper position, without moving fields or variables; in addition, as

stated before, the use of CompletableFuture widely reduces the number of instructions required to handle threads.

In [14], the authors presented an approach to analyze Java streams; their proposed tool statically analyzes a stream pipeline and verifies whether to run the stream in parallel or not. This approach is strictly related to Java streams, while our tool covers a wider set of possible optimizations, since any method call could be evaluated for parallel execution. Other examples of refactoring activities for specific instructions are available in the literature: in [13], Java arrays and their loops were refactored to ParallelArray by using anonymous classes; in [34], an optimized compiler was proposed for the automatic parallelization of loops; in [35], a refactoring tool for the X10 programming language was presented to introduce additional concurrency within loops.

In [23,24], the authors proposed two different libraries, for Java and C++, respectively. These APIs provide many features to ensure multicore parallel programming, cluster parallel programming, GPU-accelerated, and big data parallel programming. Developers can manually integrate code with these features to change it from sequential to parallel. However, one of the main difficulties that developers face when developing parallel applications is to understand which code fragment can be suitable for parallel execution [6] and which concurrent APIs could better fit a particular instruction [12]; instead, our proposal aims to solve these issues automatically. Indeed, the needed statements were selected according to an accurate analysis and a refactored parallel version generated via appropriate APIs.

Many approaches focus on the optimization of the compiled code to achieve parallelism [25–27]. These approaches automatically analyze executable code to identify coarse grain tasks, such as the iteration of a large loop, and execute them in parallel. These approaches prove that the performance gain is considerable; however, the optimization process is not visible to the developer. We propose a tool that shows the changes made to the source code, providing a clear view of which sections are selected and how they are properly refactored to achieve parallelism; moreover, the generated code is available to developers, who can add, modify, or remove it according to their will.

Asynchronous programming is widely used in Android applications, because of UI access and I/O operations [36]. In [15], a tool was proposed to automatically detect long-running operations and refactor them into asynchronous operations. Moreover, in [19], a tool was described to identify the improper use of asynchronous constructs and change them. Unlike our approach, the previously mentioned approach focused on analyzing code that was already parallel, aiming to identify defects. Moreover, JavaScript ecosystems provide synchronous and asynchronous calls to handle several I/O operations. In [16], the authors proposed a refactoring approach to assist developers in transforming operations from synchronous to asynchronous. Our approach is more comprehensive, as it is not just focused on I/O operations, which can be handled as well as other instructions. Arteca et al. [22] presented an approach to reorder asynchronous calls to be executed as early as possible, yielding significant performance benefits. For this approach, the input code is parallel, and the developer chooses the parts that can run in parallel, unlike our more automated approach.

## 3. Proposed Approach

The proposed approach uses static analysis to gather data from the source code of the analyzed software. Parsing activities are based on the JavaParser library. This is an automatic parser that takes one or more .java files as input, generates an abstract syntax tree (AST) for each one, and provides means to perform operations on ASTs, such as reading, inserting, deleting, and updating [37]. Algorithm 1 shows the pseudo-code at a high level of abstraction of our analysis. The procedure takes the source code of the analyzed application as input, and, after parsing, it returns the instances of CompilationUnit for the application. A CompilationUnit is a class modeling a single Java file represented as an AST. Afterward, the method's declarations are extracted from each CompilationUnit instance to obtain a list containing all the methods implemented in the source code. A method body generally

contains many instructions; in our analysis, we focus on method calls since they are the instructions that will be evaluated to be run in parallel among all the instructions in the method's body. Each method call is analyzed according to its context, data dependence, and the number of instructions performed. All three analyses are covered in Sections 4 and 5. Once a method call passes all the checks, it is transformed to a parallel version and the method is updated with the proper synchronization statement, according to the data-dependent statement found. Finally, all the compilation units containing at least one transformed method will be written in new Java files.

**Algorithm 1** The algorithm of the proposed approach.

```
procedure TRANSFORMSEQUENTIALTOPARALLEL(Sc)
    compilationUnits ← parseAllPaths(Sc)
    for cu, compilationUnits do
        methods ← visitMethods(cu)
    end for
    for m, methods do
        methodCalls ← m.getMethodCalls()
        for mCall, methodCalls do
            if CONTEXTANALYSIS(mCall) then
                ddStatement ← DATADEPENDENCEANALYSIS(m, mCall)
                if CFGANALYSIS(m, mCall, ddStatement) then
                    mCall ← transformToParallel(mCall, ddStatement)
                end if
            end if
        end for
    end for
    printParallelCu()
end procedure
```

## 4. Method Call Analysis

Running methods could take considerable execution times since such methods could have a large number of instructions or encapsulate nested method calls within them. To speed up the execution, some methods could be executed in a new dedicated thread. We performed three different analyses to ensure that the proposed automatic source transformation into a parallel version preserves the behavior and the correctness of the original version. The first two analyses are described in the subsections below, while the third one is discussed in Section 5 since it requires broader insight. Firstly, we analyze the context of the method call to assess that inserting a parallel construct is safe; secondly, we check data dependencies between concurrent instructions to avoid race conditions; thirdly, we evaluate the number of instructions that threads would run before synchronization is needed, to determine whether the workload is balanced between them and if the number of instructions is sufficiently large to obtain performance gain.

### *4.1. Method Context Analysis*

The context of the method call has crucial significance when evaluating whether there is a gain when running the method call in parallel. The context is the statement where the method call is retrieved; it could be the method call itself, or a more complex statement containing it. For the former case, further analysis of the same statement is not needed since the method call is the only instruction in the statement; hence, we can check further conditions to determine whether parallel execution is possible and desired. Conversely, for the latter cases, i.e., the method call is found within another statement. There could be some cases where parallel execution is unsuitable; hence, we avoid further analysis aimed at parallelization.

Algorithm 2 shows the instructions executed when having to analyze the context of a method call. The context is extracted by obtaining the ancestor of the method call.

The ancestor is the parent node on the AST of the node containing the analyzed method call. If the method call is part of a more complex statement, the ancestor will be the instruction containing it; otherwise, the ancestor will be the statement containing the block of instructions with the method call, e.g., the body of the method declaration, or the body of a for statement. Once the ancestor is retrieved, it is compared to a set of feasible contexts, which was defined beforehand. The function returns true if the context of the method call taken as input satisfies the feasible contexts, and false otherwise.

**Algorithm 2** The instructions performed by the context analysis.

**function** CONTEXTANALYSIS(mCall)
  $context \leftarrow getAncestor(mCall)$
  $suitableContexts \leftarrow defineSuitableContext()$
  **return** $suitableContexts.contains(context)$
**end function**

Listing 1 shows some examples of method calls that are unsuitable for parallel execution, i.e., method calls that are a part of the condition expression in *if*, *while*, and *do* constructs (examples 1 and 2 in Listing 1), and statements, such as *switch*, *for*, *throw*, *assert*, and *synchronized* (example 3). Parallel execution is unsuitable in such cases because the returned value of the method call is immediately used to determine whether to execute the following statements. Similarly, all the method calls that are in a return statement cannot give any advantage when running in parallel as the need for the return value would just make the calling thread wait for the result.

**Listing 1.** Method calls in contexts where parallelization is deemed unsuitable. The method call is part of (i) an *if* condition, (ii) a *while* condition, or (iii) a *for* loop iteration.

```
// Method call checkForFileAndPatternCollisions() in a condition statement
1. if (checkForFileAndPatternCollisions()) {
    addError("File property collides with fileNamePattern. Aborting.");
    addError(MORE_INFO_PREFIX + COLLISION_URL);
    return;
}
// Method call isRunning() in a while statement
2. while (isRunning() != state) {
    runningCondition.await(delay, TimeUnit.MILLISECONDS);
}

// Method call getCopyOfStatusListenerList() as iterable in a foreach statement
3. for (StatusListener sl : sm.getCopyOfStatusListenerList()) {
    if (!sl.isResetResistant()) {
        sm.remove(sl);
    }
}
```

Listing 2 shows six examples of method calls that can be transformed to run in parallel. Line 1 shows a method call used as the value for an assignment expression; line 2 shows a method call used as the value for an assignment in a variable declaration expression; line 3 shows a method call chained with other calls; line 4 shows a method call passed as the argument for other method calls; line 5 shows a method call without any other instruction; and line 6 shows a method call passed as the argument for an object creation expression. In such cases, the result of the method call is not used to determine whether to execute the following statements. Indeed, our aim is to evaluate whether the whole statement (which could consist of several expressions) could run in a thread parallel to the thread that runs the following statements.

**Listing 2.** Method calls in contexts where parallelization is deemed suitable. The method call is (i) part of an assign expression, (ii) part of a variable declaration expression, (iii) chained with other method calls, (iv) passed as the argument for a method call, (v) called without other instructions, and (vi) passed as the argument for an object creation expression.

```
// Assign expression using a method call
1. le = makeLoggingEvent(aMessage, null);)

// Variable declaration expression using a method call
2. BufferedReader in2 = gzFileToBufferedReader(file2);)

// Chained method calls
3. context.getStatusManager().getCount();

// Method call getClassName() passed as argument for another method call
4. buf.append(tp.getClassName());

// A simple method call
5. implicitModel.markAsSkipped());

// Method call passed as argument for an object creation
6. new Parser(tokenizer.tokenize());
```

### 4.2. Data Dependence Analysis

Accessing data shared among threads should be properly guarded. We used a tool proposed in one of our previous works, which provides a set of APIs to extract data dependence for methods [38]. This approach analyzes both variables and method calls inside a method to define its input set (i.e., the set of variables read) and output set (i.e., the set of variables written).

Algorithm 3 shows the instructions executed when performing the data dependence analysis. The statement of the method call is retrieved; hence, the input set and the output set of the statement are defined. Therefore, for every following statement, the intersection between sets is computed; if the intersection has at least one element, the statement will be returned since it is the data-dependent statement where the synchronization must be inserted; conversely, the step goes to the next statement. If there is no data-dependent statement, the last statement of the method is returned.

**Algorithm 3** The instructions performed by the data dependence analysis.

**function** DATADEPENDENCEANALYSIS(m, mCall, ddStatement)
  $s_1 \leftarrow getStatement(m, mCall)$
  $inputSetS_1 \leftarrow getInputSet(s_1)$
  $outputSetS_1 \leftarrow getOutputSet(s_1)$
  $index \leftarrow indexOf(m, s_1)$
  **for** $i \leftarrow index + 1, m.getStatements().length() - 1$ **do**
    $s_2 \leftarrow m.getStatements().get(i)$
    $inputSetS_2 \leftarrow getInputSet(s_2)$
    $outputSetS_2 \leftarrow getOutputSet(s_2)$
    **if** $checkIntersections(inputSetS_1, outputSetS_1, inputSetS_2, outputSetS_2)$ **then**
      **return** $s_2$
    **end if**
  **end for**
  $lastStatement \leftarrow getLastStatement(m)$
  **return** $lastStatement$
**end function**

Listing 2 shows several statements. For lines with an assignment, such as lines 1 and 2, the output set consists of the variable being assigned and the variables modified by the called method, discovered by inspecting the called method. Hence, for line 1, the output set consists of variable le, and for line 2, the output set consists of variable in2; let us suppose that no variables are written in the called methods. The input set comprises all variables

passed to the called method and the variables read within the called method. Therefore, for line 1, the input set consists of the aMessage variable, and for line 2, its input set consists of the file2 variable; let us suppose that no variables are read in the called methods.

For methods that are called on an instance, using a variable holding the reference, such as the calls at lines 3, 4, 5, and 6, the variable holding the reference is part of the input set, because when executing the method call, such a variable will be read to properly dispatch the message. Hence, variables buf and tp are the input sets for line 4, and variables implicitModel and tokenizer are the input sets for lines 5 and 6, respectively. For lines 3, 4, 5, and 6, their respective output sets will have all the variables written in the called method, along with the variables used to call the method (e.g., context for line 3, tp and buf for line 4, etc.) if the called method writes some of the attributes in the same instance. Once the said two sets are defined for the concerned statement, the analysis is repeated for every following statement to find any data dependencies.

Listing 3 shows an example of the data dependence analysis. For line 74, method getRandomlyNamedLoggerContextVO() is the one we are analyzing, and the method call is found within a variable declaration expression; hence, as stated before, we consider this instruction feasible. Then, starting from line 75, the data dependence analysis is performed to find a data-dependent statement, which is found on line 79 because variable lcVO is assigned to e.loggerContextVO field. The data-dependent statement represents the point where the main thread will have to wait for the forked thread to finish before it can continue with its execution.

**Listing 3.** Data dependence analysis: lcVO is declared at line 74 (the statement has it in its output set) and is used at line 79 (the statement has it in its input set); hence, the two statements are data-dependent as the intersection between their output and input sets is not empty.

```
74  LoggerContextVO lcVO = corpusModel.getRandomlyNamedLoggerContextVO();
75  PubLoggingEventVO[] plevoArray = new PubLoggingEventVO[n];
76  for (int i = 0; i < n; i++) {
77      PubLoggingEventVO e = new PubLoggingEventVO();
78      plevoArray[i] = e;
79      e.loggerContextVO = lcVO;
80      e.timeStamp = corpusModel.getRandomTimeStamp();
    ....
}
```

## 5. Control Flow Graph Analysis

Having found all data dependencies among statements, the analysis aims at finding the paths that could execute in parallel. Possible parallel paths are basically found by analyzing the control flow graph (CFG) [31], i.e., a representation by means of a graph of all the paths that could be executed by a program. Then, a further assessment is performed to determine whether the introduced parallelism will provide performance gains. Therefore, the last part of the proposed analysis determines if the amount of work that will be assigned to both threads is balanced and adequately large to make it worth the effort required to start a new thread. Several studies show how the number of statements is significant in evaluating the computational complexity of an algorithm [39,40]. Our approach uses the control flow graph theory to inspect the execution paths and evaluate the number of instructions that compose each path. Library JGraphT (https://jgrapht.org, accessed on 21 July 2023) was used to assist the CFG analysis needed, i.e., finding paths.

Algorithm 4 shows the instructions executed for the CFG analysis. Firstly, the CFG is built and the conditional and loop branches are checked to make the graph acyclic. Secondly, the node of the statement containing the method call that could be executed in parallel and the data-dependent statement (previously defined) are collected. Finally, the paths that could be executed in parallel are computed and compared. The function returns true if the two paths are suitable (see Section 5.3) for the parallelization, and false otherwise.

**Algorithm 4** The instructions performed by the control flow graph analysis.

**function** CFGANALYSIS(m, mCall, ddStatement)
$cfg \leftarrow$ BUILDCFG$(m)$
$cfg \leftarrow handleConditionalAndLoops(cfg)$
$node_1 \leftarrow getGraphNode(cfg, m)$
$node_2 \leftarrow getGraphNode(cfg, ddStatement)$
**return** COMPAREPATHS$(node_1, node_2)$
**end function**

### *5.1. Building the CFG*

A CFG represents the set of instructions that form a program and all the interactions that intervene. In a CFG, each node represents a basic block, i.e., a line of code that consists of a specific statement, while an edge represents a jump from one instruction to another. A CFG could contain nodes with more than one incoming or outgoing edge (e.g., conditional statements); moreover, cycles are allowed, given the presence of loop statements (e.g., *for*, *while* loops) and recursive method calls.

Listing 4 shows the source code for a method called start() and Figure 1 displays its CFG; the method is implemented in the LevelChangePropagator class from the lombok library (https://github.com/projectlombok/lombok, accessed on 17 April 2023). During the analysis, as this method contains two method calls, with their own CFGs, the CFG of start() was built with four nodes plus all the nodes of the CFGs for the called methods, named $CFG_{96}$ and $CFG_{98}$. Let us suppose that a single node is found to have more than one method call during the analysis; if so, then all the CFGs of the called methods will be connected following the order of execution.

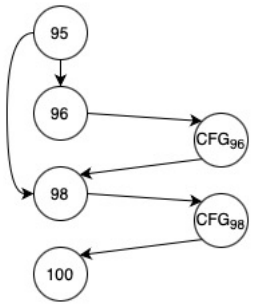


**Figure 1.** Control flow graph of the code shown in Listing 4.

**Listing 4.** Method start() has two method calls: one at line 96 and the other at line 98. The method is part of the lombok library.

```
 94  public void start() {
 95      if (resetJUL) {
 96          resetJULLevels();
 97      }
 98      propagateExistingLoggerLevels();
 99
100     isStarted = true;
101  }
```

Algorithm 5 shows the instructions executed to build a CFG, given a method declaration. Our approach builds a CFG for a method; for every method call found inside the first method, its CFG is merged with the main one (i.e., the one of the caller). The method calls identified can be traced to methods implemented in the application under analysis; hence, they are analyzed, or they could be methods implemented as libraries, which are not analyzed due to the lack of source codes. Hence, when a method's body contains a call to a library-implemented method, we represent it as a single node of the CFG. The first and last nodes are saved to ease the merging processes between CFGs.

**Algorithm 5** The algorithm used to build the control flow graph analysis for a method.

```
procedure BUILDCFG(m)
    cfg ← createEmptyCfg()
    for statement ← m.getStatements() do
        node ← createNode(statement)
        if firstNode = null then
            firstNode ← node
        end if
        cfg.addNode(node)
        if previousStmt ≠ null then
            cfg.addEdge(previousStmt, node)
        end if
        if statement.contains(MethodCall) then
            mCall ← statement.getMethodCall()
            if mCall.getCfg() = null then
                BUILDCFG(mCall)
            end if
            cfg.addEdge(node, mCall.getCfg().getFirstNode())
            previousStmt ← mCall.getCfg().getLastNode()
            if statement.next() = null then
                lastNode ← previousStmt
            end if
        else
            previousStmt ← node
        end if
        if statement.next() = null then
            lastNode ← node
        end if
    end for
end procedure
```

### 5.2. Handling Conditional Branches and Loops

To evaluate the workload that will be assigned to each thread, it is necessary that the execution paths are clearly defined and there is no ambiguity. Hence, given a pair of nodes in the graph, just one path should exist connecting them. This assumption must be satisfied by all nodes in the CFG. To ensure that, we need to properly handle conditional branches and loops. For conditional statements, having two alternative paths, it would be necessary to prune one branch from the graph. We follow the worst-case execution time (WCET) approach [41], i.e., we select the branch with the highest number of instructions, as this is likely the one with the lengthiest execution time, while the other branch is disconnected from the graph. Therefore, pruned nodes are not connected to any other node; however, we keep these nodes in our representation because they could be reconnected to the graph when needed for further analysis.

Listing 5 shows an example where a conditional statement has a method call in each branch. When the analysis decides to parallelize the method call within the *then* branch (i.e., line 79) because of the WCET, the *else* branch will be removed from the graph; the resulting CFG is shown on the left side of Figure 2. Otherwise, the *then* branch will be removed; see the right side of Figure 2. Using the example in Listing 4, despite there being no *else* branch, the corresponding CFG will be modified by removing the edge connecting node 95 to node 98, because branch 95→96→$CFG_{96}$→98 is longer (i.e., with more instructions, and higher WCET value) than branch 95→98.

Regarding loop management, several approaches statically estimate the impact of the loops in the program execution [42,43]. In our approach, to keep the analysis fast and effective, we consider loops as simple block statements, and we analyze them without estimating how many cycles could be run (such an analysis is outside the scope of the

proposal, although loop estimation results could be readily included, without affecting the generality of our proposed approach). However, we keep track of these occurrences when evaluating the workload of the branch.

Finally, another case should be considered: a chain of method calls that presents cycles, which means a method could call a previous method in the chain, creating a loop that, when inserting the corresponding CFGs in the main one, can break the assumption of one path for every pair of nodes. We observe that these occurrences are very rare, and if found, we simply remove the edge that creates the loop in the graph.

**Listing 5.** A code fragment of an if/else statement with a method call (apply(...)) in both blocks. Both method calls are in the context of a variable declaration expression (lines 79 and 81).

```
78  if (ruleEntry.type == RuleEntry.TYPE_TEST_RULE) {
79      result = ((TestRule) ruleEntry.rule).apply(result, description);
80  } else {
81      result = ((MethodRule) ruleEntry.rule).apply(result, method, target);
82  }
83  return result;
```

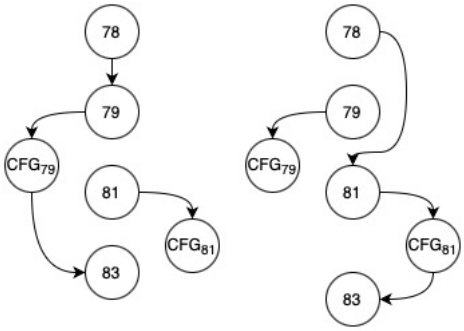


**Figure 2.** Control flow graphs of the code with conditions shown in Listing 5.

### *5.3. Define and Evaluate the Two Parallel Paths*

The last step of our approach consists of evaluating the instructions that compose the two paths, each of which could be assigned to a thread. The first path will be from the node containing the instruction we want to run in parallel and the following instruction, while the other path is from the latter to the data-dependent statement. Algorithm 6 shows the instructions executed by the approach. The paths are computed as Dijkstra shortest path algorithm, using the API provided by the third-party library JGraphT. By construction, the shortest path is the only path between two given nodes. The length of a path represents the number of nodes (instructions) within it; therefore, we check the lengths of both paths by filtering out paths with short lengths and paths with differences between their lengths that are bigger than a threshold. The function returns true if the paths satisfy these two requirements, and false otherwise.

Figure 1 shows the CFG of the method in Listing 4; node 96 should be executed in parallel, while node 100 is the data-dependent one; the first path is from node 96 to node 98, while the second path is from node 98 to node 100. The length of the first path is 17, while the length of the second path is 42 (this is the count of the instructions constituting the path that considers the instructions of the called methods). When the main thread reaches the instruction at line 100, it will wait until the other thread is finished. Since both paths have a sufficiently large number of instructions, the method call is refactored to execute in parallel.

Listing 6 shows the code of the method refactored for parallel execution by using CompletableFuture, as it allows running the method at line 96 asynchronously (using runasync()). The instruction at line 99, future.join(), is the waiting point for the main thread until the task defined at line 96 finishes.

**Algorithm 6** The algorithm used to define and compare two parallel paths.

**function** COMPAREPATHS($node_1, node_2$)
  $m \leftarrow node_1.getStatement().getMethodCall()$
  $cfg_m \leftarrow m.getCfg()$
  $lastNode \leftarrow cfg_m.getLastNode()$
  $nextNode \leftarrow lastNode.getNext()$
  $path_1 \leftarrow DijkstraShortestPath.findPathBetween(node_1, lastNode)$
  $path_2 \leftarrow DijkstraShortestPath.findPathBetween(nextNode, node_2)$
  **if** $path_1.getLength() \geq 5 \wedge path_2.getLength() \geq 5$ **then**
    **if** $|path1.length() - path_2.length()| < 10$ **then**
      **return** *true*
    **end if**
  **end if**
  **return** *false*
**end function**

**Listing 6.** Method start() updated with CompletableFuture, executing the method called at line 96, where variable future is initialized, and for line 99, where future.join() waits for its completion.

```
    public void start() {
94      CompletableFuture<Void> future;
95      if (resetJUL) {
96          future = CompletableFuture.runAsync(() -> resetJULLevels());
97      }
98      propagateExistingLoggerLevels();
99      future.join();
100     isStarted = true;
    }
```

## 6. Experiments and Results

Experiments were aimed at automatically analyzing and modifying an application, while assessing the correctness of the results and the performance gains. We tested our approach on a sample application that extracted data from the Amazon Books Reviews dataset [44] to draw inferences on books, authors, and reviews. The code of the analyzed application is available in a public repository (https://github.com/AleMidolo/BookReviews, accessed on 21 July 2023). The dataset contains about 3$M$ book reviews for 212,404 unique books and many users who provided reviews for books. We selected a subset of 166,667 reviews for the analysis to have a reasonable execution time for testing the correctness of the transformation and evaluating the performance.

### *6.1. Code Analysis and Transformation*

Among the 44 methods in the six classes of the application under analysis, our tool automatically identified five methods that were refactored to a parallel version accordingly. Listing 7 shows one of such methods, getUserForAuthor(HashMap<String, Book> books, List<Review> reviews), which has been automatically analyzed as follows.

Method call getAuthors() at line 2 (Listing 7) was selected; firstly, the context was such that further assessment was undertaken, as the context of the method call was one of the feasible cases detailed in Section 4.1; the data dependence analysis identified the instruction at line 4 as data-dependent, because the authors list was filled by the method call at line 2 and then was used at line 4. Then, the CFG was built (composed of statements in the called methods) and conditional branches and loops were analyzed to possibly perform some adjustments (see Section 5.2); the code was free from any if/else statement; hence, the CFG had no alternative paths. Moreover, the CFG had no loops. Then, possible parallel paths were found. Finally, the path comprising line 2 (including the CFG of the called method), and the path comprising line 3 (including the CFG of the called method) were

automatically assessed to count the number of instructions constituting them, and then determine if their parallel execution would provide performance gains.

**Listing 7.** Extracting the authors from the books and the users from the reviews; then assigning to each author all the users that provided at least one review for the author's books.

```
   public static HashMap<String, Author> getUserForAuthor(HashMap<String, Book> books,
       List<Review> reviews) {
1       ExtractData extractor = new ExtractData(books, reviews);
2       HashMap<String, Author>  authors = extractor.getAuthors();
3       HashMap<String, User> users = extractor.getUserForAuthor();
4       authors.values().forEach(author -> {
5               List<String> usersId = author.getBooks().stream().
6                       flatMap(b -> b.getReviews().stream().map(r -> r.getUserID())).
7                       collect(Collectors.toList());
8               usersId.forEach(u -> author.addUser(users.get(u)));
9       });
10      return authors;
   }

// parallel version of some instructions above
2   CompletableFuture<HashMap<String, Author>>  f = CompletableFuture.supplyAsync(() ->
     extractor.getAuthors());
3   HashMap<String, User> users = extractor.getUserForAuthor();
4   HashMap<String, Author>  authors = f.get();
```

Figure 3 shows a portion of the CFG for the code in Listing 7, from line 1 to line 5. The paths that should be executed in parallel are (i) the instruction at line 2; (ii) the instruction at line 3. Line 4 presents a statement that depends on the output of line 2. The first path has nine instructions, eight are the instructions contained in the $CFG_2$, of which, six are inside a loop. The second path has 10 instructions, 11 are the instructions contained in the $CFG_3$, of which, 5 are inside a loop. The method is suitable for parallel refactoring because the instruction numbers are sufficiently large and both CFGs found contain a loop.

Therefore, a new version of the method has been generated that contains the constructs for parallel execution. Such a version can be seen in the bottom part of Listing 7; the method call was given as a lambda expression in the CompletableFuture call supplyAsync(); just after line 3, the get() call was introduced to synchronize the execution and create the authors HashMap.

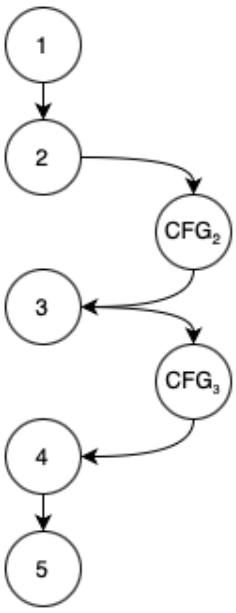


**Figure 3.** CFG extracted from method getUserForAuthor() shown in Listing 7.

### *6.2. Measured Execution Times*

We assessed the performance of the generated refactored parallel version by using Java Microbenchmark Harness (JMH), a standard performance test harness that provides APIs to write formal performance tests.

We configured the library to have warm-up cycles for performance measurements; each run had five warm-up iterations and five normal iterations. JMH tests are the best indicators of performance improvements since they isolate the subject that is evaluated, avoiding influences by other subjects.

Table 1 shows the resulting benchmarks of the methods that were automatically refactored by our approach. Method column presents the name of the refactored method; Time (ms) column displays the execution time in milliseconds for the method; Speed-Up column is the speed-up obtained by the parallel version (computed as $runtime_{old}/runtime_{new}$); finally, the sub-columns Sequential and Parallel show the respective measured execution times for the sequential ($runtime_{old}$) and parallel ($runtime_{new}$) versions. The average overhead measured for each CompletableFuture call was 7.77 ms, calculated on 10 different runs and after 10 warm-up runs. The overhead represents the time needed for the JVM to create and execute the new thread that will handle the parallel path; it was computed as the difference between times for the execution of the parallel and sequential versions. The measured performance will be further discussed in Section 7.

**Table 1.** Execution times for all five methods refactored for the sequential and parallel executions.

| **Method** | **Time (ms)** | | **Speed-Up** |
|---|---|---|---|
| | **Sequential** | **Parallel** | |
| extractFromDataset | 36 K | 17 K | 2.11 |
| extractMostReviewedAuthor | 485 | 330 | 1.46 |
| extractLeastReviewedAuthor | 522 | 318 | 1.64 |
| extractAverageReviewedAuthor | 533 | 317 | 1.68 |
| getUserForAuthor | 700 | 486 | 1.44 |

### *6.3. Correctness Evaluation*

To prove the validity of our analysis and transformation for the case study, we implemented a test suite that checked whether the generated parallel version kept the same behavior as the sequential one. The JUnit framework was used to write and execute all tests. Table 2 displays the test implemented for the five refactored methods; Description column provides a brief description of what the test checks; Result column shows the value returned by both sequential and parallel executions. Every test has an assertion where the expected value is the output given by the sequential execution of the method, while the actual value is the output of the parallel one.

For every test that was executed, the result given by the parallel version was the same as the one given by the sequential version. For getUserForAuthor() method, the only one whose return value was a HashMap, all the values in the HashMap were the same in both executed versions. The same goes for books and reviews extracted by the extractFromDataset() method.

**Table 2.** Tests executed to validate the correctness of the refactored parallel code.

| Description | Called Method | Result |
|---|---|---|
| number of books | extractFromDataset() | 212,404 |
| number of reviews | extractFromDataset() | 333,335 |
| all books are equal | extractFromDataset() | true |
| all reviews are equal | extractFromDataset() | true |
| author with highest number of reviews | extractMostReviewedAuthor() | Lois Lowry |
| maximum number of reviews for author | extractMostReviewedAuthor() | 3822 |
| number of books for most reviewed author | extractMostReviewedAuthor() | 8 |
| author having the lowest number of reviews | extractLeastReviewedAuthor() | John Carver |
| minimum number of reviews for an author | extractLeastReviewedAuthor() | 0 |
| number of books for the least reviewed author | extractLeastReviewedAuthor() | 1 |
| author with the average review number | extractAvgReviewedAuthor() | Mark Bando |
| average number of reviews for an author | extractAvgReviewedAuthor() | 14 |
| number of books for the author with the average number of reviews | extractAvgReviewedAuthor() | 2 |
| number of authors | getUserForAuthor() | 127,279 |
| all authors are equal | getUserForAuthor() | true |
| all users are equal | getUserForAuthor() | true |

## 7. Discussion

### *7.1. Performance Gain*

To properly evaluate the effectiveness of the approach, a performance benchmark was executed to measure the actual gain in terms of the execution time, and a suite of tests was run to check the correctness of the transformations performed.

The average speed-up was 1.66 for all five methods benchmarked, from the highest one, extractFromDataset(), with 2.11, to the lowest one, getUserForAuthor(), with 1.44. Of course, the performance of the parallel version and, hence, the obtained speed-up, is mostly affected by the number of operations executed by the two parallel paths. When there is only a slight difference between the execution time of each path, then there is a maximum gain, since the paths run in parallel, without having one waiting for the other. As discussed in Section 5, the analysis built a CFG to estimate the number of operations for each path. We identified the actual number of operations and whether there were loops within them, but not an estimate of the execution time; hence, there could be some differences between the execution times of the two paths. This factor might negatively influence the performance gain since the path with less work will have to wait for the other one to complete.

The five benchmarks executed for the application have shown varying speed-ups (see Table 1. For extractFromDataset() method, the two parallel threads had very similar execution times, and the performance gain was high. Conversely, for the methods with 1.44 and 1.46 speed-ups, there was a greater difference between the execution times of the two parallel paths. Despite this, the results have shown that our transformations were effective since they provided significant performance gains. The extractFromDataset() benchmark required a longer execution time compared to the other benchmarks because it performed several I/O operations to read the data from the .csv files, which, as presented in Section 6, consisted of more than 200,000 records for books and 300,000 records for reviews.

Table 2 provides insight into the tests executed to check the correctness of our applied refactoring. Result column presents the values returned by both sequential and parallel versions, showing that—in every execution—the results were unchanged. This shows that the control flow and data dependence analysis (Section 4.2) were correctly performed to identify dependencies, and that synchronization statements were introduced in the proper positions. A misidentification could have led to desynchronization and, thus, test failure.

We checked the code coverage of our test suite with JaCoCo (https://www.eclemma.org/jacoco/, accessed on 21 July 2023), an open-source library for Java that automatically performs code coverage analyses. The total code coverage of our test suite was 94%, as only the catch branches for try statements were not executed. All the branches that

were sequentially and parallelly executed were covered, proving the correctness of the transformation. The output of the code coverage analysis of the analyzed application is available on the GitHub repository.

The overhead measured represents approximately 0.05% of the parallel execution time for extractFromDataset() method, and between 1.5% and 2.5% for the other methods, minimally affecting parallel executions.

### 7.2. Validity Treats

Our approach aims at ensuring the correctness of transformation and performance improvements by minimizing the potential overhead of parallel threads. As the approach is based on the static analysis of code, there are some scenarios that cannot be evaluated.

Firstly, when subsequent instructions, or blocks of instructions, are found to have some data dependence, and their needed estimated computational efforts are low, then we do not transform the code into a parallel one. On the one hand, this prevents cluttering the code with instructions that start parallel execution and perform synchronization when performance gain is uncertain. However, on the other hand, the analysis only estimates the computational efforts of instructions, and inaccuracies of estimates could occur, e.g., when there are cycles, because the executed repetition numbers are unknown, when there are calls to methods provided by external libraries, etc. For this, our tool might not gather all potential performance gains. The estimation of execution time could be improved, and some indications given by the developer could be considered.

Secondly, the static analysis alone cannot guess the method that will be executed at runtime when the analyzed code uses polymorphism. In the case of polymorphism, we assume the worst case, and consider the less favorable data dependence among methods and the less favorable performance gains. Therefore, in some cases, we could miss opportunities for parallel execution.

Thirdly, when a fragment of code uses several references to objects belonging to the same type, by statically analyzing the code, it is very difficult to distinguish the different instances, e.g., when a variable is conditionally assigned one among several references. We miss the opportunity to have parallel execution for distinct objects and further opportunities for performance gains.

Finally, another possible area for performance gain that could be overlooked by our approach is when instruction reordering is possible, i.e., the subsequent dependent statements could be moved far away from each other while preserving correctness.

The above limitations have been embedded in our analysis and tool to ensure that when executing the transformed code, the behavior is correct, although at the expense of performance in a few cases.

## 8. Conclusions

This paper presents an approach and a corresponding tool that statically analyze the source code of an application, to identify opportunities for parallel execution. The approach is based on the analysis of the control flow and data dependence to assess the possibility of executing some statements in parallel while preserving the program's correctness. For the proposed approach, the control flow graph of an application under analysis is automatically obtained; this graph is used to find possible parallel paths and estimate potential performance gains.

A transformed parallel version of the application can then be automatically generated to have methods that execute in a new thread by using a Java CompletableFuture. The changes to the source code are minimal, making the transformation clean and effective.

We performed many experiments to test the correctness and efficacy of the approach and the corresponding developed tool. The resulting parallel version automatically executes in a fraction of the time taken by the sequential version. Moreover, the correctness of the transformation was assessed by executing several tests. The test suite employed covered

94% of the code, and when executed, showed that the transformations performed kept the results of the executed methods unchanged.

The produced tool can be very useful for updating legacy applications and supporting the development of new applications.

**Author Contributions:** Conceptualization, E.T.; methodology, A.M. and E.T.; software, A.M.; validation, A.M. and E.T.; writing—original draft preparation, A.M. and E.T.; writing—review and editing, A.M and E.T.; supervision, E.T. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Data Availability Statement:** Data used for the experiments and the code of the analyzed application can be found at https://github.com/AleMidolo/BookReviews.

**Acknowledgments:** This research was supported by the University of Catania Piaceri 2020/22 Project "TEAMS".

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Yang, C.; Lu, T.; Yan, S.; Zhang, J.; Yu, X. N-Trans: Parallel Detection Algorithm for DGA Domain Names. *Future Internet* **2022**, *14*, 209. [CrossRef]
2. Murugappan, M.; Thomas, J.V.J.; Fiore, U.; Jinila, Y.B.; Radhakrishnan, S. COVIDNet: Implementing Parallel Architecture on Sound and Image for High Efficacy. *Future Internet* **2021**, *13*, 269. [CrossRef]
3. Iqbal, U.; Abosekeen, A.; Georgy, J.; Umar, A.; Noureldin, A.; Korenberg, M.J. Implementation of Parallel Cascade Identification at Various Phases for Integrated Navigation System. *Future Internet* **2021**, *13*, 191. [CrossRef]
4. Kaddoura, S.; Haraty, R.A.; Al Kontar, K.; Alfandi, O. A Parallelized Database Damage Assessment Approach after Cyberattack for Healthcare Systems. *Future Internet* **2021**, *13*, 90. [CrossRef]
5. Herlihy, M.; Shavit, N.; Luchangco, V.; Spear, M. *The art of Multiprocessor Programming*; Morgan Kaufmann: Cambridge, MA, USA, 2020.
6. Ahmed, S.; Bagherzadeh, M. What Do Concurrency Developers Ask about? A Large-Scale Study Using Stack Overflow. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Oulu, Finland, 11–12 October 2018.
7. Pinto, G.; Torres, W.; Castor, F. A Study on the Most Popular Questions about Concurrent Programming. In Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools, Pittsburgh, PA, USA, 26 October 2015; pp. 39–46. [CrossRef]
8. Pinto, G.; Torres, W.; Fernandes, B.; Castor, F.; Barros, R.S. A large-scale study on the usage of Java's concurrent programming constructs. *J. Syst. Softw.* **2015**, *106*, 59–81. [CrossRef]
9. Fox, G.C.; Williams, R.D.; Messina, P.C. *Parallel Computing Works!*; Morgan Kaufmann: Cambridge, MA, USA, 2014.
10. Zhang, Y.; Li, L.; Zhang, D. A survey of concurrency-oriented refactoring. *Concurr. Eng.* **2020**, *28*, 319–330. [CrossRef]
11. Ishizaki, K.; Daijavad, S.; Nakatani, T. Refactoring Java Programs Using Concurrent Libraries. In Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD), Toronto, ON, Canada, 17 July 2011; pp. 35–44. [CrossRef]
12. Dig, D.; Marrero, J.; Ernst, M.D. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE), Vancouver, BC, Canada, 16–24 May 2009; pp. 397–407. [CrossRef]
13. Dig, D.; Tarce, M.; Radoi, C.; Minea, M.; Johnson, R. Relooper: Refactoring for Loop Parallelism in Java. In Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA), Orlando, FL, USA, 25–29 October 2009; pp. 793–794. [CrossRef]
14. Khatchadourian, R.; Tang, Y.; Bagherzadeh, M. Safe automated refactoring for intelligent parallelization of Java 8 streams. *Sci. Comput. Program.* **2020**, *195*, 102476. [CrossRef]
15. Lin, Y.; Radoi, C.; Dig, D. Retrofitting concurrency for android applications through refactoring. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), Hong Kong, China, 16–21 November 2014; pp. 341–352. [CrossRef]
16. Gokhale, S.; Turcotte, A.; Tip, F. Automatic Migration from Synchronous to Asynchronous JavaScript APIs. *Proc. ACM Program. Lang.* **2021**, *5*, 1–27. [CrossRef]
17. Wloka, J.; Sridharan, M.; Tip, F. Refactoring for reentrancy. In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Amsterdam, The Netherlands, 24–28 August 2009; pp. 173–182. [CrossRef]

18. Schäfer, M.; Sridharan, M.; Dolby, J.; Tip, F. Refactoring Java Programs for Flexible Locking. In Proceedings of the 33rd International Conference on Software Engineering (ICSE), Honolulu, HI, USA, 21–28 May 2011; pp. 71–80. [CrossRef]
19. Ozkan, B.K.; Emmi, M.; Tasiran, S. Systematic asynchrony bug exploration for android apps. In Proceedings of the International Conference on Computer Aided Verification, LNCS 9206, San Francisco, CA, USA, 18–24 July 2015; Springer: Berlin/Heidelberg, Germany; pp. 455–461. [CrossRef]
20. Zhang, Y.; Shao, S.; Liu, H.; Qiu, J.; Zhang, D.; Zhang, G. Refactoring Java Programs for Customizable Locks Based on Bytecode Transformation. *IEEE Access* **2019**, *7*, 66292–66303. [CrossRef]
21. Zhang, Y.; Shao, S.; Zhai, J.; Ma, S. FineLock: Automatically refactoring coarse-grained locks into fine-grained locks. In Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA), Virtual Event, USA, 18–22 July 2020; pp. 565–568. [CrossRef]
22. Arteca, E.; Tip, F.; Schäfer, M. Enabling Additional Parallelism in Asynchronous JavaScript Applications. In Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP), Virtual Event, Aarhus, Denmark, 11–17 July 2021. [CrossRef]
23. Kaminsky, A. Parallel Java Library. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16–21 November 2014.
24. Haidl, M.; Gorlatch, S. PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14. In Proceedings of the LLVM Compiler Infrastructure in HPC (LLVM-HPC), New Orleans, LA, USA, 17 November 2014; pp. 1–11. [CrossRef]
25. Kimura, K.; Taguchi, G.; Kasahara, H. Accelerating Multicore Architecture Simulation Using Application Profile. In Proceedings of the 10th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC), Lyon, France, 21–23 September 2016; pp. 177–184. [CrossRef]
26. Kasahara, H.; Kimura, K.; Adhi, B.A.; Hosokawa, Y.; Kishimoto, Y.; Mase, M. Multicore Cache Coherence Control by a Parallelizing Compiler. In Proceedings of the 41st IEEE Annual Computer Software and Applications Conference (COMPSAC), Torino, Italy, 4–8 July 2017; Volume 1, pp. 492–497. [CrossRef]
27. Felber, P.A. Semi-automatic Parallelization of Java Applications. In Proceedings of the on the Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE, Catania, Italy, 3–7 November 2003; Springer: Berlin/Heidelberg, Germany, 2003; pp. 1369–1383. [CrossRef]
28. Bernstein, A.J. Analysis of Programs for Parallel Processing. *IEEE Trans. Electron. Comput.* **1966**, *EC-15*, 757–763. [CrossRef]
29. Maydan, D.E.; Hennessy, J.L.; Lam, M.S. Efficient and Exact Data Dependence Analysis. *SIGPLAN Not.* **1991**, *26*, 1–14. [CrossRef]
30. Wolfe, M.; Banerjee, U. Data Dependence and Its Application to Parallel Processing. *Int. J. Parallel Program.* **1987**, *16*, 137–178. [CrossRef]
31. Allen, F.E. Control flow analysis. *ACM Sigplan Notices* **1970**, *5*, 1–19. [CrossRef]
32. Dig, D.; Marrero, J.; Ernst, M.D. How Do Programs Become More Concurrent: A Story of Program Transformations. In Proceedings of the 4th ACM International Workshop on Multicore Software Engineering (IWMSE), Honolulu, HI, USA, 21 May 2011; pp. 43–50. [CrossRef]
33. Zhang, Y. Improving the learning of parallel programming using software refactoring. *Comput. Appl. Eng. Educ.* **2017**, *25*, 112–119. [CrossRef]
34. Larsen, P.; Ladelsky, R.; Lidman, J.; McKee, S.A.; Karlsson, S.; Zaks, A. Parallelizing more Loops with Compiler Guided Refactoring. In Proceedings of the 2012 41st International Conference on Parallel Processing (ICPP), Pittsburgh, PA, USA, 10–13 September 2012; pp. 410–419. [CrossRef]
35. Markstrum, S.A.; Fuhrer, R.M.; Millstein, T.D. Towards concurrency refactoring for x10. *A Sigplan Not.* **2009**, *44*, 303–304. [CrossRef]
36. Dig, D. Refactoring for Asynchronous Execution on Mobile Devices. *IEEE Softw.* **2015**, *32*, 52–61. [CrossRef]
37. Smith, N.; Van Bruggen, D.; Tomassetti, F. *Javaparser: Visited*; Leanpub, oct. de: Victoria, BC, Canada, 2017. Available online: https://scholar.google.com.tw/citations?view_op=view_citation&hl=zh-TW&user=9u0kf8UAAAAJ&citation_for_view=9u0kf8UAAAAJ:HDshCWvjkbEC (accessed on 1 August 2023).
38. Midolo, A.; Tramontana, E. An API for Analysing and Classifying Data Dependence in View of Parallelism. In Proceedings of the ACM International Conference on Computer and Communications Management (ICCCM), Okayama, Japan, 29–31 July 2022; pp. 61–67. [CrossRef]
39. Gulwani, S.; Mehra, K.K.; Chilimbi, T. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. *SIGPLAN Not.* **2009**, *44*, 127–139. [CrossRef]
40. Sinn, M.; Zuleger, F.; Veith, H. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In Proceedings of the International Conference on Computer Aided Verification (CAV), Vienna, Austria, 18–22 July 2014; Springer International Publishing: Cham, Switzerland, 2014; pp. 745–761. [CrossRef]
41. Gustafsson, J.; Betts, A.; Ermedahl, A.; Lisper, B. The Mälardalen WCET benchmarks: Past, present and future. In Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET), Brussels, Belgium, 6 July 2010; Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik: Wadern, Germany, 2010. [CrossRef]

42. Lokuciejewski, P.; Marwedel, P. Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization. In Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS), Dublin, Ireland, 1–3 July 2009; pp. 35–44. [CrossRef]
43. Lokuciejewski, P.; Cordes, D.; Falk, H.; Marwedel, P. A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models. In Proceedings of the International Symposium on Code Generation and Optimization (CGO), Seattle, WA, USA, 22–25 March 2009; pp. 136–146. [CrossRef]
44. Amazon Book Reviews Dataset. Available online: https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews (accessed on 21 July 2023).

**Disclaimer/Publisher's Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.