Information Retrieval-based Fault Localization for Concurrent Programs

Shuai Shao
Department of Computer Science and Engineering
University of Connecticut
Storrs, CT, USA
shuai.shao@uconn.edu

Tingting Yu
Department of Computer Science and Engineering
University of Connecticut
Storrs, CT, USA
tingting.yu@uconn.edu

***Abstract*—Information retrieval-based fault localization (IRFL) techniques have been proposed as a solution to identify the files that are likely to contain faults that are root causes of failures reported by users. These techniques have been extensively studied to accurately rank source files, however, none of the existing approaches have focused on the specific case of concurrent programs. This is a critical issue since concurrency bugs are notoriously difficult to identify. To address this problem, this paper presents a novel approach called BLCoiR, which aims to reformulate bug report queries to more accurately localize source files related to concurrency bugs. The key idea of BLCoiR is based on a novel knowledge graph (KG), which represents the domain entities extracted from the concurrency bug reports and their semantic relations. The KG is then transformed into the IR query to perform fault localization. BLCoiR leverages natural language processing (NLP) and concept modeling techniques to construct the knowledge graph. Specifically, NLP techniques are used to extract relevant entities from the bug reports, such as the word entities related to concurrency constructs. These entities are then linked together based on their semantic relationships, forming the KG. We have conducted an empirical study on 692 concurrency bug reports from 44 real-world applications. The results show that BLCoiR outperforms existing IRFL techniques in terms of accuracy and efficiency in localizing concurrency bugs. BLCoiR demonstrates effectiveness of using a knowledge graph to model the domain entities and their relationships, providing a promising direction for future research in this area.**

***Index Terms*—Concurrent program, fault localization, information retrieval**

I. Introduction

Concurrency bugs are prolific in deployed concurrent systems, leading to frequent failures. To reduce manual effort involved in debugging concurrency bugs, researchers have proposed various techniques to automatically localize bugs [1]–[10]. These techniques aim to identify faulty program elements related to software failures. Two categories of fault localization techniques exist: spectrum-based and information retrieval (IR)-based [3], [11]–[14]. Spectrum-based techniques analyze dynamic information from program executions to localize faults, but require both passing and failing executions and can cause significant runtime overhead.

Information retrieval-based fault localization (IRFL) has been proposed as an alternative to spectrum-based techniques, eliminating the need for dynamic information. IRFL identifies files that likely contain the root causes of reported failures without requiring passing and failing executions. Bug reports are used as queries to compute the similarity between each source file and the report. The ranked source files indicate their relevancy to the bug report. The greater the similarity between the source file and the bug report, the more likely the source file is responsible for the failures. IR-based techniques have been shown to be equally effective as spectrum-based fault localization techniques [15].

Although IRFL has been extensively studied with the goal of accurately ranking source files, *none of the existing approaches have focused on concurrent programs*. The unique challenge is that existing techniques often use simple NLP (e.g., tokenization, stemming, etc) to process the query in the bug reports and thus lack domain information related to concurrency bugs. Some approaches have been proposed to improve queries by designing filters [16], [17]. However, concurrency bugs have unique semantics, in which the buggy code often involves synchronization operations and shared resource accesses. Existing queries often lack such structured information that is unique to concurrency semantics and thus may not be effective in localizing concurrency bugs.

In this paper, we present a novel approach, named BLCoiR, which utilizes IR-based fault localization techniques to identify the root cause of failures described in bug reports. Our method involves representing the information contained in a bug report as a domain-specific knowledge graph (KG), which comprises domain concepts related to concurrent programming features and their semantic relationships. Since bug reports typically contain unstructured text, we employ natural language processing (NLP) techniques to extract relevant domain concepts pertaining to concurrency from the reports and use them to construct the KG. The KG serves as a basis for formulating queries for fault localization using information retrieval. By leveraging the graph structure, we can use ranking algorithms (e.g., Page Rank) to identify the likely source files leading to the reported concurrency bug. To the best of our knowledge, BLCoiR is the first approach to localize concurrency bugs from bug reports with information retrieval.

We evaluate BLCoiR on four open datasets from 44 projects consisting of 692 concurrency bug reports by using the widely used performance metrics. First, we evaluate BLCoiR against

the baseline method. The results show that BLCoiR localizes concurrency bugs with 37.19% higher accuracy (Top 10), 55.87% higher result ranks (MRR), and 57.42% higher precision (MAP) than the baseline (Section IV-B1). Second, we evaluate BLCoiR's query component by comparing it with the baseline queries, including queries generated by GPT-3.5. The results show that BLCoiR localizes concurrency bugs with 23%-34% higher accuracy (Top 10), 14%-39% higher ranks (MRR), and 14%-40% higher precision (MAP) than the baseline queries (Section IV-B2). Third, we compare BLCoiR with six state-of-the-art IRFL techniques. The results show that our approach could have an equivalent performance by only considering unstructured information from bug reports. Then we combine with state-of-the-art techniques (i.e., stack trace analysis, source files analysis) to improve BLCoiR and we have BLCoiR+. BLCoiR+ performs a 8%-28% higher accuracy (Top 10), 11%-40% higher ranks (MRR), and 12%-44% higher precision (MAP) than state-of-the-art IRFL techniques (Section IV-B3).

In summary, this paper makes the following contributions:

- We propose a novel knowledge graph-based approach for performing IR-based fault localization, which involves representing the information contained in bug reports as a domain-specific knowledge graph.
- We introduce BLCoiR, the first automated tool designed specifically for localizing buggy source files from bug reports related to concurrency bugs.
- We implemented BLCoiR and conducted an empirical study to evaluate its effectiveness and efficiency on real-world bug reports.

II. Problem Statement and Motivation

One of the open challenges in IRFL is the quality of queries. The specific quality concerns involve lack of relevant information and too much irrelevant information. As a result, the query does not return the correct buggy code or artifacts (e.g., source files). There has been some work on studying different ideas to reformulate queries for improving the performance of IRFL [17]–[23]. While existing techniques address the quality concerns of queries in a variety of ways, most of them do not take into account the structure of the NL sentences or perform well with bug reports containing excessive noises. For instance, although Rahman et al. [19] propose a context-aware approach to classify the quality of bug report queries (i.e., noisy, rich, poor) and reformulate the queries by using graph-based term weighting with appropriate keywords, this method does not effectively consider domain-specific entities and their importance, thereby limiting its ability to localize relevant bugs, such as concurrency bugs. In summary, the primary limitation of existing techniques is that they treat words in natural language descriptions equally, without effectively determining the significance of domain-specific entities. Consider the following bug report from Apache CAMEL[1], as shown in Table I.

[1]https://issues.apache.org/jira/browse/CAMEL-7715

TABLE I: A Bug Report from CAMEL.

Bug ID	CAMEL-7715
Summary	SjmsConsumer and SjmsProducer do not remove thread pool when stop
Description	SjmsConsumer and SjmsProducer always register a new ThreadPool on Camel context ExecutorServiceManager every time a new instance is created for an endpoint. If consumer or producer is stopped or removed or even component is removed, thread pool still exists.

TABLE II: Example Queries.

Technique	Query	Rank
BLIZZARD	sjms consumer producer sjmsproducer pool thread instance stop sjmsconsumer context time sjmsconsumer threadpool executorservicemanager service camel manager thread executor endpoint pool register remove created stop producer component consumer removed stopped exists	15th
Manual	SjmsConsumer SjmsProducer remove ThreadPool Camel context ExecutorServiceManager instance endpoint	1st

This bug report describes a concurrency-related issue about SjmsConsumer and SjmsProducer invoking ThreadPool. To generate a search query, we used BLIZZARD [19], a state-of-the-art query-based IRFL technique. The resulting query, shown in Table II, contained noise words like "stop", "time", and "exist," which caused the correct result to appear at a low-rank position (15th). Since the bug is concurrency-related, we hypothesized that sentences containing concurrency-related API (i.e., ThreadPool) and other code entities (i.e., SjmsConsumer) may be more important for localizing the buggy code. To test this hypothesis, we manually selected important keywords from the summary and the first sentence of the description and used them as the query. This manual query returned the correct result at the first position. These results indicate that to precisely localize the root cause of a reported bug, reformulating the query by considering important domain-specific entities may be necessary.

The Solution. The key idea of our approach is formulating the queries based on graph theoretic computations to capture the importance of the terms. Specifically, we employ a knowledge graph [24], which has proven to be a useful tool for representing and managing knowledge. The knowledge graph can accurately convey complex semantic information across various types of data by representing entities as nodes and their relationships as edges. We believe that knowledge graph (KG) provides a unique opportunity to address challenges in localizing domain-specific bugs (e.g., concurrency bugs) with information retrieval. Since KG is intended to represent information of specific entities and their logic relations, it is appropriate for representing the concurrency entities mentioned in the bug report texts and their semantic relationships. By organizing these concepts into a graph structure, we can distinguish the importance of different domain entities in the text description and improve query formulation accordingly.

Comparing to Existing Work. There has been some work on using graph theory in IR [25], in which vertices typically represent IR objects such as keywords, documents, authors, and/or citations, and in which bidirectional links represent their weighted associations or relevance. Graph representation has advantages over traditional text processing as it can preserve

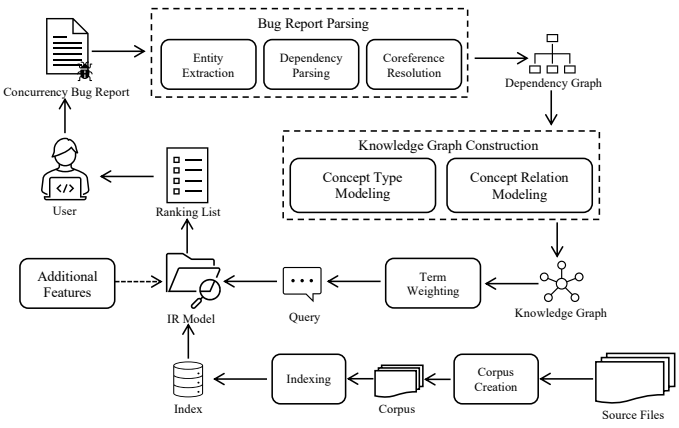


Fig. 1: The overview of BLCoiR.

structured entities and their relationships while eliminating noise in the query. There have been some approaches on using graph theory to perform IR activities to accomplish software engineering tasks. For example, Rahman et al. [19] develop a text graph to identify important terms from a textual body using co-occurrences and syntactic dependencies among the words and then use such terms to formulate IR queries. However, none of these techniques have focused on formulating queries by identifying domain entities or their semantic relationships, which limits their ability to precisely localize the root cause of concurrency faults.

III. Approach

Figure 1 shows the overview of BLCoiR, which contains two major steps: Bug Report Parsing and Knowledge Graph Construction. The Bug Report Parsing component takes the natural language description of a bug report as input. It then employs several natural language processing techniques to build a dependency graph to represent the grammatical structure of the bug report. Next, the Knowledge Graph Construction develops a knowledge graph to model the concurrency entities and their semantic relationships based on the dependency graph. The knowledge graph is used to construct the IR query to localize buggy files from the project source by using the IR models. The output of BLCoiR is a ranked list of source files, showing which files are most likely to have the concurrency bug.

A. Corpus Creation and Indexing

Given the source files, our process commences with the execution of lexical analysis and the construction of the corpus for information retrieval (Line 2, Algorithm 1). In this endeavor, we employ CoreNLP [26] to execute various commonly utilized NLP techniques, encompassing word tokenization, the elimination of stopwords (such as '(', ')', 'a', 'the', etc.), and the exclusion of keywords pertinent to programming languages (such as 'int', 'double', 'char', etc.). To disentangle concatenated words (e.g., TypeDeclaration) into distinct entities (i.e., "type" and "declaration"), we deploy concatenation word splitting. Additionally, CoreNLP facilitates lemmatization, which aligns words with their respective lemmas. For instance, words such as "gets" and "getting" are mapped to the lemma "get". Upon the corpus's establishment, each source file corresponds to a collection of words, collectively constituting the corpus itself. Ultimately, we generate an index encompassing the entire corpus, associating terms with their respective positions within the documents (Line 3, Algorithm 1). This indexing procedure empowers the system to pinpoint files containing words from a given query and subsequently rank these files based on their relevance.

B. Bug Report Parsing

Bug report parsing aims to extract word entities and their dependencies, which are used to construct knowledge graphs. Our approach involves analyzing the grammatical structure of bug report sentences to identify these entities and their dependencies. Using this information, we construct a knowledge graph that highlights the word entities related to concurrency bugs, and their relationships based on the semantics of concurrent programs. By leveraging the dependencies among the word entities, we can derive insights that help to understand concurrency bugs.

Key Entity extraction. We first use Stanford CoreNLP [26] to split each report into sentences and apply Part-Of-Speech (POS) tagging to each sentence. To identify words relevant to the source code, we use Java Naming Conventions [27] as a guide and only consider nouns and verbs for the conceptual graph. We group all the verbs into a set called VERB and all the nouns into a set called NOUN (Line 4, Algorithm 1). We then categorize the two sets into different concept types (as described in Section III-C) to build the knowledge graph. By leveraging this approach, we can generate a structured and organized representation of the relevant information contained within issue reports.

Sentence Dependency Parsing. To comprehend the meaning of a sentence, we identify the relationships between the tagged word entities to determine the sentence's grammatical structure. The resulting dependency graph is then utilized to construct the knowledge graph for formulating the query. CoreNLP's dependency parser [26] is used to parse the sentence and assign semantic roles like subject and direct object (Line 6, Algorithm 1). The semantic dependency graph of the sentence "*Deadlock if using separate thread to write to ServletOutputStream*" is depicted in Figure 2, where the graph's edges represent the dependencies between two words such as attribute, subject, object, etc. For instance, the word `write` has an object in `ServletOutputStream` and a subject in `thread`. Consequently, there is an edge from `write` to `ServletOutputStream` and another edge from `write` to `thread`.

Coreference Resolution. To accurately identify the relevant word entities in a sentence, it is necessary to perform coreference resolution to identify all expressions that refer to the same entity in a text. For example, consider the following two

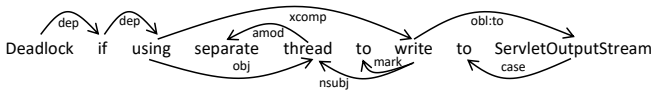


Fig. 2: An Example of Dependency Graph.

sentences from an Apache CAMEL bug report [2]: "*XsltBuilder uses a ResultHandler instance variable to hold the result of the transformation. That makes its thread unsafe and at the same time...*". In this example, coreference resolution analysis helps to identify that the possessive adjective `its` in the second sentence refers to the `XsltBuilder` in the first sentence. We employ CoreNLP's coreference resolution system [26] to conduct coreference resolution (Line 8, Algorithm 1). Subsequently, we integrate the coreference resolution chains with the dependency graph (Line 9, Algorithm 1). By using coreference resolution, we can ensure that words with coreference share the same node in the knowledge graph, making it possible to accurately capture the relationships between entities. Therefore, if two words have coreference, they will be represented by the same node in the knowledge graph.

Algorithm 1 The algorithm of BLCoiR

1: **procedure** BLCoiR(D_{BR}, D_{SF}) ▷ D_{BR}:bug reports
2: $Corpus \leftarrow CorpusCreation(D_{SF})$ ▷ D_{SF}:source files
3: $Index \leftarrow CorpusIndexing(Corpus)$
4: $Entity \leftarrow KeyEntityExtraction(D_{BR})$
5: /* DenGraph: semantic dependency graph */
6: $DenGraph \leftarrow SemanticDependency(D_{BR})$
7: /* Coref: coreference resolution chain */
8: $Coref \leftarrow CoreferenceResolution(D_{BR})$
9: $DenGraph \leftarrow Coref \cup DenGraph$
10: /* Concept type and relation modeling */
11: $CT, CR \leftarrow ConceptModeling(Entity)$
12: **while** $node_i \in DenGraph$ **do**
13: **if** $node_i \notin CT$ **then** ▷ CT: concept types
14: /* in(): return incoming edges from a node */
15: **while** $node_j \in node_i.in()$ **do**
16: /* out(): return outgoing edges from a node */
17: $node_j.out() \leftarrow node_i.out()$
18: **end while**
19: remove($node_i$)
20: **end if**
21: **end while**
22: **while** $cr_k \in CR$ **do** ▷ CR: concept relations
23: $KG \leftarrow KnowledgeGraph(DenGraph, cr_k)$
24: **end while**
25: $Query \leftarrow PageRank(KG)$
26: **return** $RetrievalModel(Index, Query)$
27: **end procedure**

C. Knowledge Graph Construction

Although bug reports often contain a wealth of information related to a reported issue, the specific knowledge that we focus on is the natural language description of the bug. By extracting the bug information and representing it as a knowledge graph, we can capture the domain concepts related to concurrency bugs and how these concepts interact with each other. Our approach involves identifying the terms that are most relevant to concurrency bugs and mapping them to the relevant source code. To accomplish this, we have developed nine types of concurrency entities and nine types of relations that describe the semantic relationships between these entities. The generalizability of these concepts and relationships are discussed in Section V.

Concept Type Modeling. After applying natural language processing to all the concurrency bug reports, we end up with two sets of words: VERBs and NOUNs. We classify these words into nine different concept types based on their meanings in the context of the sentences. These concept types are presented in Table III.

For nouns, we divide them into six different concept types. The first type is Concurrency Bug (CBG), which consists of terms used to describe various concurrency faults such as deadlock and livelock. The second type is Concurrency Mechanism (CME), which includes programming mechanisms related to concurrency, such as locks and threads. Synonym for Bug (SYB) is the third type, which covers the words that indicate faults, such as bug, issue, and problem. The fourth type is Code Entity (COE), which refers to code entities like class and method names, and the fifth type is Exceptions (EXC), which comprises the names of exceptions like NullPointerException. Finally, the sixth type is Other Nouns (OTN), which encompasses all other nouns that do not fall under the first five categories.

For verbs, we classify them into three different concept types. The first type is Concurrency Operation (COP), which represents programming operations that relate to concurrency programming, such as lock and fork. The second type is Concurrency Symptom (CSY), which includes programming symptoms that usually describe concurrency faults, such as hang and block. The third type is Other Verbs (OTV), which covers all other verbs that do not fall under the first two categories. We categorize these concept types based on our experience and by analyzing various concurrency-related sentences. The reason for considering words categorized as Other Verbs (OTV) and Other Nouns (OTN), which are not directly related to concurrency bugs, is that as previously mentioned, nouns and verbs can have relevance to source code naming. Therefore, we retain these words, allowing the knowledge graph, through term weighting methods, to discern their significance. This approach enables the determination of their importance within the context.

Once we have identified these concept types, we proceed to iterate through all nodes in the dependency graph. We remove nodes that do not belong to any concept type and replace their edges by redirecting their predecessors to their successors (Lines 12-21, Algorithm 1).

Concept Relation Modeling. The relations between concepts play a critical role in constructing the knowledge graph that

[2]https://issues.apache.org/jira/browse/CAMEL-140

TABLE III: Concept Types

Abbr	Concept Types	Words	POS
CBG	Concurrency Bug	deadlock, livelock, race, etc.	NOUN
CME	Concurrency Mechanism	lock, thread, writelock, etc.	NOUN
SYB	Synonym for Bug	bug, issue, problem, error, etc.	NOUN
COE	Code Entity	lock(), unlock(), etc.	NOUN
EXC	Exceptions	NullPointerException, etc.	NOUN
OTN	Other Nouns	servlet, web, server, etc	NOUN
COP	Concurrency Operation	lock, fork, release, etc.	VERB
CSY	Concurrency Symptom	hang, block, freeze, etc.	VERB
OTV	Other Verbs	have, make, take, etc	VERB

TABLE IV: Concept Relations

ID	Semantic Patterns	Meanings
CR1	COE+CBG	Concurrency bugs in entities
CR2	COE+(SYB|EXC|CSY)	Bugs in entities
CR3	CME+CBG	Concurrency bugs in concurrency mechanisms
CR4	CME+(SYB|EXC|CSY)	Bugs in concurrency mechanisms
CR5	COP+CBG	Concurrency bugs in concurrency operations
CR6	COP+(SYB|EXC|CSY)	Bugs in concurrency operations
CR7	COP+OTN$_{obj}$	Concurrency operation on objects
CR8	OTN$_{nsubj}$+CBG$_{obj}$	Potential concurrency bugs
CR9	CR1+CR3+CR5	Concurrency-related items stand out

capture the meaning of a concurrency bug and extract key information. To model these relations, we define seven concept relations, as shown in Table IV.

CR1, CR3, and CR5 capture the relations between CBG and COE, CME, and COP, respectively, representing the semantic meaning of concurrency bugs that occur in certain code entities, concurrency mechanisms, or concurrency operations. Similarly, the relations between SYB, EXC, and CSY with COE, CME, and COP represent the semantic meaning of bugs/faults that occur in certain code entities, concurrency mechanisms, or concurrency operations. We additionally consider grammar dependency for CR7 and CR8. For CR7, OTN will have a concept relation with COP if OTN is the *object* of COP, indicating a concurrency operation (COP) on an object (OTN) based on the grammar dependency. For CR8, OTN will have a concept relation with CBG if OTN is the *subject* and CBG is the *object*, indicating that OTN may have a concurrency bug (CBG) based on the grammar dependency. The purpose of pattern CR9 is to strengthen the weight of concurrency-related concept nodes, i.e., they will gain more weight during the term weighting process.

These concept relations represent the meaning of concurrency-related sentences. Concepts that have relations with others will have a link, and by traversing each concept relation on the dependency graph, we obtain a concurrency bug report specific knowledge graph (Lines 23, Algorithm 1).

Figure 3 depicts the knowledge graph transformed from the dependency graph in Figure 2. The blue edge between *Deadlock* and *ServletOutputStream* is generated based on the semantic pattern CR1, representing the meaning that there is a concurrency bug (deadlock) in a code entity (ServletOutputStream). The red edges are generated based on the semantic pattern CR9, which strengthens the weight of concurrency-related concept nodes.

D. Term Weighting

After converting the bug report text to a graph, we apply the PageRank algorithm [28] to detect the most important concept

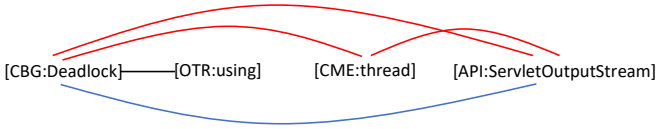


Fig. 3: An Example of Knowledge Graph.

nodes. PageRank, which Google initially used to rank web pages in search results, evaluates each node's significance in the graph based on its connections and weights. It has been applied effectively in various fields, such as citation analysis and social networks. The PageRank algorithm determines the importance of each concept node in a conceptual graph by evaluating its connectivity with other nodes, including their respective weights. The formula for PageRank is given by:

$$S(V_i) = (1-d) + d * \sum_{j \in In(V_i)} \frac{1}{|Out(V_j)|} S(V_j)$$

Here, $S(V_i)$ represents the PageRank score for a given vertex V_i. $In(V_i)$ is the set of vertices that point to V_i, and $Out(V_j)$ is the set of vertices that vertex V_j points to. The damping factor d has a value between 0 and 1, with 0.85 being the commonly adopted value in recent works.

We use an undirected graph, ensuring that the out-degree of a vertex is equal to its in-degree, to apply PageRank (Line 25, Algorithm 1). After running PageRank, we sort the nodes by their score in descending order and select the top L (1<L<30, see Figure 5) nodes to form the query. These top-ranking nodes represent the salient concepts in the bug report and are considered the most important for retrieval purposes.

E. Retrieval and Ranking

The retrieval and ranking of relevant buggy files is accomplished by analyzing textual similarity between the query and each file in the corpus. Several models exist for information retrieval, such as VSM [29], SUM [15], LSI [30], and LDA [31]. VSM and its variants, like rVSM [32], are the most commonly used in IR. Previous research evaluated the effectiveness of rVSM with other models, including SUM, LSI, and LDA, showing that rVSM outperforms them [32]. However, manually building and evaluating IR models is time-consuming, and state-of-the-art IRFL techniques use existing retrieval tools like Indri and Lucene. Thus, rVSM, Indri [33], and Lucene [34] are selected as candidate IR models for this study, and Section IV-B1 evaluates these models to determine the best one for BLCoiR. Lastly, we feed the generated query and indexed source files into the IR model, which subsequently provides a ranking list for the source files (Line 26, Algorithm 1).

IV. Evaluation

We will answer the following research questions.

RQ1: How effective is BLCoiR in localizing concurrency bugs? In this research question, we investigate the overall

effectiveness of BLCoiR in accurately ranking source files that contain concurrency bugs. Additionally, we evaluate the impact of different retrieval models on the performance of BLCoiR.

RQ2: How does the query of BLCoiR compare to the base queries? This research question aims to compare the results of BLCoiR with those obtained using baseline queries. Moreover, we examine how the query length of BLCoiR affects its performance.

RQ3: Does our approach outperform existing IRFL techniques? The objective of this research question is to compare the effectiveness of our proposed approach with existing IRFL techniques in localizing concurrency bugs.

A. Experimental Setup

1) Dataset Collection: We have gathered a comprehensive dataset consisting of 57,414 bug reports from 44 open-source projects. Out of this dataset, we identified 692 concurrency-related bug reports, which are listed in Table V. Our dataset includes projects from three popularly used sources, as well as one dataset that we collected ourselves.

Dataset$_{Bli}$: This dataset was used by the state-of-the-art IRFL work BLIZZARD [19]. The dataset includes 5,139 bug reports and 28,544 source files collected from six projects, namely ECF, Eclipse JDT, Eclipse PDE, and Tomcat. We examined all the bug reports in this dataset and identified 170 concurrency-related bug reports. **Dataset**$_{LR}$: This dataset was used in Ye's work LR [35] and contains 22,747 bug reports collected from six projects, namely AspectJ, Birt, Eclipse JDT, Eclipse Platform UI, SWT, and Tomcat, during the period from 2001 to 2014. However, the bug reports of Eclipse JDT and Tomcat overlapped with Dataset$_{Bli}$. Therefore, we identified 50 concurrency-related bug reports from the remaining four projects in this dataset. **Dataset**$_{Ben}$: This dataset was collected by Lee et al. [36] and comprises 46 projects from Apache, Commons, JBoss, Spring, and Wildfly, with a total of 9,459 bug reports. However, not all of these projects had concurrency bugs. After analyzing the dataset, we identified 278 concurrency-related bug reports from 26 of these projects. **Dataset**$_{Git}$: This dataset is a collection of eight projects that we selected from GitHub. Our selection process was based on the following criteria: 1) the projects were widely used in concurrency-related works, such as high-performance projects, distributed systems, and parallel programming; 2) the projects were still active and constantly changing; 3) since concurrency bugs have a lower occurrence rate in open-source software, we selected projects that had a significant number of bug reports. After analyzing the data, we identified a total of 201 concurrency-related bug reports from the eight selected projects, which had a total of 27,399 bug reports.

Establishing ground-truths. To accurately identify and analyze concurrency bugs, it is crucial to have access to the source files containing these bugs. These files serve as ground-truths, enabling researchers to compare and verify the accuracy of their findings. The original datasets, namely Dataset$_{Bli}$[3], Dataset$_{LR}$[4], and Dataset$_{Ben}$[5], were provided by their respective authors and are available on GitHub. To construct the ground-truths, we need to link each reported bug with its corresponding source files. For Dataset$_{Git}$, the fixed bug reports come with commit information that displays the changes made to the files in the process of fixing the bugs. In this case, we can use the modified files as ground-truths. This approach ensures that our analysis is based on accurate and reliable data.

Identifying concurrency bug reports. To collect reports of concurrency bugs, we employed a filtering approach. Firstly, we applied commonly used keywords related to concurrency such as "thread", "race", "deadlock", and "order violation" to all bug reports as suggested by previous studies [37]–[39]. However, as some bug reports may describe faults unrelated to concurrency using these keywords, we manually inspected each report to determine its relevance to concurrency. This manual filtering enabled us to eliminate irrelevant reports and collect 692 reports of concurrency bugs.

Our dataset's size is comparable to those used in existing IRFL research [19], [32], [35], [36], [40]–[43]. These datasets' sizes range from 111 to 22,747 bug reports, while a case study on concurrency bugs in open-source software by Asadollah et al. [39] indicated that approximately 4% of open-source software bugs are related to concurrency issues.

Quality of bug reports. The effectiveness of identifying faults in a bug report depends heavily on the types of data included in it, as mentioned in a study cited in the source [19]. The study has two categories of bug reports: those that include program entities in their textual descriptions, such as class names and file names, and those that only use natural language without any program entities. The former is considered rich information and is classified as Dataset$_{PE}$ (i.e., high-quality bug reports), while the latter is considered poor information and is classified as Dataset$_{NL}$ (i.e., poor-quality bug reports) For instance, a bug report that includes program entities like SjmsConsumer and SjmsProducer, as seen in Table I, falls under Dataset$_{PE}$. The number of bug reports classified in Dataset$_{PE}$ was 549, while Dataset$_{NL}$ contained 143 bug reports, as detailed in Table V. This classification of bug reports can help in analyzing the impact of different types of data on fault localization, ultimately leading to better bug identification and resolution.

2) Baseline Techniques.: Our approach to fault localization relies heavily on the effectiveness of our query reformulation component. To evaluate the performance of this component, we designed five baseline queries: 1) **BR** considers the entire content of a bug report, including stack traces; 2) **BRText** only considers the text description of a bug report; 3) **Concept** uses only the extracted entities as queries to evaluate the usefulness of Concept Relation Modeling (Section III-C); 4) **DenGraph** applies PageRank on only the dependency graph

[3]https://github.com/masud-technope/BLIZZARD-Replication-Package-ESEC-FSE2018

[4]https://github.com/LTR4L/ltr4l

[5]https://github.com/exatoa/Bench4BL

TABLE V: Datasets.

Dataset$_{Bli}$					**Dataset**$_{Ben}$										**Dataset**$_{Git}$				
Projects	**#BR**	**#NL**	**#PE**	**#Files**	**Projects**	**#BR**	**#NL**	**#PE**	**#Files**	**Projects**	**#BR**	**#NL**	**#PE**	**#Files**	**Projects**	**#BR**	**#NL**	**#PE**	**#Files**
ecf	18	3	15	2802	Camel	64	11	53	14522	DATAJPA	3	0	3	599	Druid	31	10	21	6542
eclipse.jdt.core	7	3	4	5908	HBase	38	8	30	3473	DATAMONGO	4	1	3	348	gRPC-Java	31	1	30	1284
eclipse.jdt.debug	32	17	15	1532	Hive	42	14	28	4953	DATAREDIS	1	0	1	318	Presto	19	4	15	7966
eclipse.jdt.ui	20	7	13	10927	Codec	1	0	1	56	SEC	15	0	15	1394	Pulsar	33	5	28	3291
eclipse.pde.ui	5	4	1	5534	Collections	1	0	1	417	SECOAUTH	1	0	1	405	Redisson	40	10	30	2160
Tomcat70	81	10	71	1841	Compress	3	0	3	233	SGF	2	1	1	328	RocketMQ	4	0	4	1581
Total	163	44	119	28544	Configuration	2	0	2	195	SHDP	1	0	1	826	Trino	10	1	9	8914
Dataset$_{LR}$					IO	5	0	5	136	SPR	1	0	1	4202	Vert.x	33	7	26	929
Projects	**#BR**	**#NL**	**#PE**	**#Files**	Lang	6	1	5	247	SWF	2	0	2	789	*Total*	201	38	163	32667
AspectJ	8	4	4	4097	Math	4	1	3	1617	SWS	3	1	2	858	**Dataset**$_{All}$				
Birt	6	0	6	8734	AMQP	17	2	15	387	WFCORE	7	1	6	3367	**Projects**	**#BR**	**#NL**	**#PE**	**#Files**
Platform	31	5	26	5053	Batch	18	2	16	1716	WFLY	32	6	26	7283	*Total*	692	143	549	131069
SWT	5	2	3	2605	DATACMNS	4	1	3	552	*Total*	278	50	228	49369					
Total	50	11	39	20489	DATAGRAPH	1	0	1	148										

TABLE VI: Comparison of IRFL Techniques.

Technique	Bug Report		External Features			IR Model
	BRT	**ST**	**VH**	**SB**	**SF**	
BugLocator	•			•		rVSM
BLUiR	•			•	•	Indri
BRTracer	•	•		•	•	rVSM
AmaLgam	•		•	•	•	Mixed
BLIA	•	•	•	•	•	rVSM
BLIZZARD	•	•			•	Lucene
BLCoiR	•					Lucene
BLCoiR+	•	•		•	•	Lucene

BRT: Bug Report Text, **ST**: Stack Trace, **VH**: Version History, **SF**: Source File, **SB**: Similar Bug Report

through the dependency parsing phase to evaluate the effectiveness and usefulness of the knowledge graph construction (Section III-C); 5) **GPT-3.5**, an advanced language model developed by OpenAI [44], to generate a query for IR based fault localization.

3) State-of-the-Art IRFL Techniques.: In order to evaluate the effectiveness of BLCoiR, we compare it with six widely adopted state-of-the-art IRFL (Information Retrieval-based Bug Localization) techniques. Table VI provides an overview of the settings for each technique. Each of these techniques considers different resources that are generally classified into two categories: Bug Report and External Features. The Bug Report resources include text descriptions (BRT) and stack traces (ST), while the External Features resources include version history (VH), similar bug reports (SB), and source files (SF). Specifically, **BugLocator** [32] suggests relevant files by examining similar bugs using rVSM. **BLUiR** [40] distinguishes between summary and description in bug reports and uses different query and document representations to perform separate searches. **BRTracer** [41] extends BugLocator using segmentation and stack-trace analysis to rank files. **AmaLgam** [42] uses three score components (version history, similar report, and structure) to rank files. **BLIA** [43] adapts the approaches of BugLocator, BLUiR, and BRTracer while using Google's algorithm for version history analysis. **BLIZZARD** [19] transforms bug report text, stack traces, and source code into graphs and uses graph-based term weighting and Lucene for fault localization.

4) Performance Metrics: We use three performance metrics to evaluate BLCoiR and other bug localization techniques. These metrics are widely used in the field and are also used by state-of-the-art techniques [19], [32], [35], [36], [40]–[43].

Top@K: This metric measures the probability that the bug localization method will successfully locate the relevant bug reports when reviewing the first k files (where $k = 1, 5, 10$) in the recommendation list generated by the method. The definition is as follows:

$$Top@K = \frac{|R_k|}{n}$$

Here, $|R_k|$ is the total number of bug reports that are successfully located when the method is recommended by Top$K$. n is the total number of bug reports used in the evaluation process.

MRR: Mean Reciprocal Rank (MRR) [45] measures the position of the first source file related to the bug report located by the bug localization method in the recommendation list. The definition is as follows:

$$MRR = \frac{1}{n}\sum_{j=1}^{n}\frac{1}{rank_j}$$

Here, $rank_j$ represents the ranking position of the first source files related to the j-th bug report in the recommendation list.

MAP: Mean Average Precision (MAP) [46] measures the average position of all source files related to the bug report located by the bug localization method in the recommendation list. The definition is as follows:

$$MAP = \frac{1}{n}\sum_{j=1}^{n} AvgP_j,\ AvgP_j = \frac{1}{|K_j|}\sum_{k\in K_j}\{Prec@k\}$$

$$Prec@k = \frac{\sum_{i=1}^{k} IsRelevant(k)}{k}$$

Here, $AvgP_j$ is the average precision for the j-th bug report, and $|K_j|$ is the total number of relevant source files for the j-th bug report. $Prec@k$ represents the precision of the top k files in the recommendation list, and $IsRelevant(i)$ returns 1 if the i-th file in the recommendation list is related to the

TABLE VII: Comparing with the Baseline.

Dataset	Technique	TOP1	TOP5	TOP10	MRR	MAP
$Dataset_{NL}$	BR	30(21.13%)	52(36.62%)	60(42.25%)	0.2757	0.1970
	BLCoiR	**45(31.69%)**	**68(47.89%)**	**77(54.23%)**	**0.3923**	**0.2597**
$Dataset_{PE}$	BR	179(32.55%)	275(50.00%)	321(58.36%)	0.4001	0.2854
	BLCoiR	**315(57.27%)**	**435(79.09%)**	**461(83.82%)**	**0.6610**	**0.4997**
$Dataset_{All}$	BR	209(26.84%)	327(43.31%)	381(50.31%)	0.3379	0.2412
	BLCoiR	**360(44.48%)**	**503(63.49%)**	**538(69.02%)**	**0.5267**	**0.3797**

bug report, and 0 otherwise. K_j is the true positive result set of a query. The higher value of each metric represents better performance.

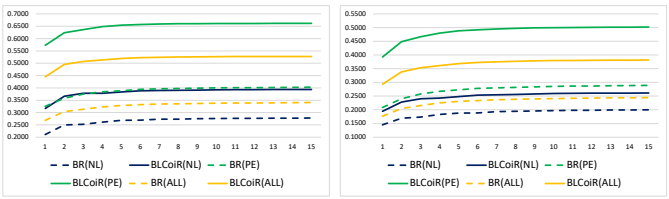
B. Results

1) RQ1: Overall Effectiveness of BLCoiR: We evaluated BLCoiR on all 44 selected projects and 692 concurrency bug reports. Lucene was used as the default information retrieval (IR) model, and we will later compare its performance with other IR models.

Table VII presents the results of BLCoiR, where the technique with the best performance on each metric is in bold. The best performance marked with "*" indicates that the technique is not statistically distinguishable from the others by using the Mann-Whitney U test [47]. As the results show, on average, BLCoiR achieved 44.48%, 63.49%, and 69.02% on the Top 1, Top 5, and Top 10 metrics, respectively, significantly outperforming the baseline (BR) by 65.72%, 46.59%, and 37.19%. The average mean reciprocal rank was 0.5267, and the mean average precision was 0.3797, which were significantly better than the baseline (BR) by 55.87% and 57.42%, respectively.

BLCoiR significantly outperforms all five metrics on $Dataset_{NL}$ and $Dataset_{PE}$ compared to the baseline techniques. Figure 4 shows BLCoiR and Baseline on different top K (K ranges from 1 to 15) values for MRR and MAP. Both techniques could reach high MRR and MAP scores after K=2. Especially, BLCoiR has the best performance on high-quality bug reports (i.e., $Dataset_{PE}$), which has significantly high MRR and MAP on K=10, 0.6610 and 0.4997, respectively. The baseline(BR) shows poor performance, with MRR and MAP scores of 0.4001 and 0.2854. The results indicate our approach can effectively select the important words and ignore the noisy words on the high-quality bug reports. However, BLCoiR has a relatively poor performance on poor-quality bug reports (i.e., $Dataset_{NL}$) compared to its performance on $Dataset_{PE}$. The baseline(BR) also suffers from very low performance, which is 0.2757 and 0.1970 on MRR and MAP. The main reason is that useful information is less in poor-quality bug reports, which means that important words are already rare to be selected by our approach. Improving the performance on poor-quality bug reports is also a challenging task for state-of-the-art techniques (discussed in RQ3).

Retrieval Models. In Section III-E, we outlined the three IR models - rVSM, Indri, and Lucene - that we considered to combine with our query. The performance of BLCoiR based on different IR models is presented in Table VIII. We observe that Lucene achieves the best performance, outperforming rVSM and Indri across all five metrics. This finding aligns with Rahman et al.'s work [19] in which they compared Indri and Lucene and found that Lucene has better performance. Consequently, we select Lucene as the IR model for BLCoiR, and all subsequent evaluations of BLCoiR are based on Lucene.



(a) Different K for MRR. (b) Different K for MAP.

Fig. 4: Comparison of BLCoiR and Baseline

TABLE VIII: Comparison of IR Models.

IR Model	TOP1	TOP5	TOP10	MRR	MAP
rVSM	42.44%	59.01%	62.78%	0.4964	0.3752
Indri	26.11%	46.65%	54.53%	0.3503	0.2629
Lucene	**44.48%**	**63.49%**	**69.02%**	**0.5267**	**0.3797**

RQ1 summary: On average, BLCoiR outperforms the baseline (BR) by 55.87% and 57.42% in terms of MRR and MAP, respectively, across all datasets. The approach effectively selects important words and ignores noisy words on high-quality bug reports, but has a relatively poor performance on poor-quality bug reports. Lucene outperforms rVSM and Indri, and is selected as the IR model for BLCoiR.

2) RQ2: Effectiveness BLCoiR's Query Component: We conducted an evaluation of BLCoiR on all datasets using four baseline queries. Table IX presents the results of BLCoiR compared to baseline queries on $Dataset_{NL}$ and $Dataset_{PE}$. Our evaluation showed that BLCoiR significantly outperformed all four baseline techniques across five metrics on average.

The BRText technique lacks the ability to consider different weights between different words, while the Entity and DenGraph techniques do consider different weights between words, but they lack domain knowledge to pinpoint bugs accurately. The queries generated by GPT-3.5 are similar to the Entity baseline, where they both select entities from bug reports. However, GPT-3.5 may select some entities that are not defined in our Entity baseline and are not helpful for bug localization. As a result, GPT-3.5 performs worse than the Entity baseline on $Dataset_{PE}$. However, it is possible to enhance GPT-3.5's performance through fine-tuning, which is a part of our future work.

Our results indicate that our approach is effective in identifying important information through conceptual graphs that incorporate domain knowledge. In particular, BLCoiR provides a 14% higher MRR and 14% higher MAP than Entity, a 39% higher MRR and 40% higher MAP than DenGraph, and a 38% higher MRR and 38% higher MAP than GPT-3.5. These results suggest that our approach achieves good performance in bug report text analysis.

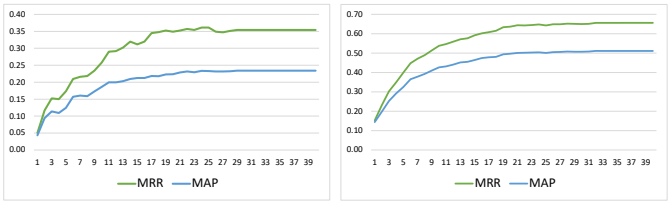
TABLE IX: Comparing with Baseline Techniques.

Dataset	Technique	TOP1	TOP5	TOP10	MRR	MAP
$Dataset_{NL}$	BRText	29(20.42%)	51(35.92%)	60(42.25%)	0.2689	0.1922
	Entity	28(19.72%)	50(35.21%)	63(44.37%)	0.2688	0.1870
	DenGraph	27(19.01%)	55(38.73%)	66(46.48%)	0.2670	0.1723
	GPT-3.5	29(20.42%)	51(35.92%)	64(45.07%)	0.2720	0.1863
	BLCoiR	**45(31.69%)**	**68(47.89%)**	**77(54.23%)**	**0.3923**	**0.2597**
$Dataset_{PE}$	BRText	192(34.91%)	290(52.73%)	335(60.91%)	0.4241	0.3068
	Entity	202(36.73%)	326(59.27%)	375(68.18%)	0.4620	0.3339
	DenGraph	158(28.73%)	274(49.82%)	323(58.73%)	0.3794	0.2718
	GPT-3.5	163(29.64%)	277(50.36%)	322(58.55%)	0.3829	0.2750
	BLCoiR	**315(57.27%)**	**435(79.09%)**	**461(83.82%)**	**0.6610**	**0.4997**
$Dataset_{All}$	BRText	221(27.67%)	341(44.32%)	395(51.58%)	0.4241	0.3068
	Entity	230(28.22%)	376(47.24%)	438(56.27%)	0.4620	0.3339
	DenGraph	185(23.87%)	329(44.28%)	389(52.60%)	0.3794	0.2718
	GPT-3.5	192(25.03%)	328(43.14%)	386(51.81%)	0.3829	0.2750
	BLCoiR	**360(44.48%)**	**503(63.49%)**	**538(69.02%)**	**0.5267**	**0.3797**

On $Dataset_{PE}$, our approach has 52% higher MRR and 61% higher MAP than each baseline technique on average. For $Dataset_{NL}$, each technique suffered poor performance, but our approach still outperformed each baseline technique by 45% higher MRR and 46% higher MAP on average.

Query Length. Figure 5 presents the results of BLCoiR query's performance on different query lengths L. The results show that the query achieves the highest MRR score on $Dataset_{NL}$ when the length is 24. However, the score of MAP is not the highest. The best performance and stability on $Dataset_{NL}$ are achieved when the query length $L > 27$. Similarly, the best performance and stability on $Dataset_{PE}$ are achieved when $L > 31$.

Based on our analysis of the results, we adopted a query length of $L = 32$ for BLCoiR. This query length achieved the best performance on both datasets, with high MRR and MAP scores. Therefore, this length is optimal for BLCoiR query's length.



(a) Different L on $Dataset_{NL}$. (b) Different L on $Dataset_{PE}$.

Fig. 5: Impact of different query length.

RQ2 summary: BLCoiR outperforms all baseline techniques with higher MRR and MAP scores. It effectively identifies important information in bug reports using a knowledge graph with domain knowledge. GPT-3.5-generated queries perform similarly to the Entity baseline but can select irrelevant entities, leading to lower performance on $Dataset_{PE}$. BLCoiR query's performance is dependent on the query length, with the optimal length being $L = 32$ for both datasets, emphasizing the importance of selecting an appropriate query length for optimal performance and stability.

3) RQ3: Comparing with the State-of-the-Art IRFL techniques: We conducted an evaluation of BLCoiR by comparing it with six state-of-the-art IRFL techniques on each dataset. To ensure fairness in the evaluation, we used the same configuration and parameters specified in each paper for all the techniques. Since different IRFL techniques have their own methods to analyze the information from stack traces, we split the bug reports in $Dataset_{PE}$ that contained stack traces into a separate dataset called $Dataset_{PES}$, and the bug reports in $Dataset_{NL}$ that contained stack traces into a separate dataset called $Dataset_{NLS}$. In total, $Dataset_{NLS}$ had 59 bug reports and $Dataset_{PES}$ had 147 bug reports.

Table X presents the evaluation results of BLCoiR in comparison to six state-of-the-art IRFL techniques on different bug report categories. Our approach exhibits the best performance on $Dataset_{PE}$, achieving an MRR of 0.6788 and a MAP of 0.5109, which is significantly better than the other techniques. However, on $Dataset_{NL}$, all state-of-the-art techniques, including BLCoiR, suffer from low performance, highlighting the difficulty of IRFL techniques in this category. Among the existing techniques, BLUiR performs slightly better with a higher MRR and MAP than others. On $Dataset_{NLS}$ and $Dataset_{PES}$, BRTracer and BLIA exhibit comparatively better MRR and MAP than BLCoiR. This is because our approach only considers the bug report text and ignores the useful information from the stack traces.

On average, BLCoiR attains a Top 1 result of 44.48%, Top 5 result of 63.49%, Top 10 result of 69.02%, an MRR of 0.5267, and a MAP of 0.3797, which is comparatively higher on all five metrics than the other six state-of-the-art IRFL techniques. However, BLCoiR does not significantly outperform BRTracer and BLIA, which utilize additional information such as analyzing similar bug reports, stack trace information, and source files, to improve their performance. While BLCoiR only considers the bug report text, the results suggest that our approach can achieve an equivalent performance with BRTracer and BLIA, without utilizing external resources for concurrency bug localization, and significantly outperform other state-of-the-art IRFL techniques.

To improve and further evaluate our approach, we decide to analyze external resources by using the methods in state-of-the-art techniques. On $Dataset_{NL}$, we found that BLUiR has a comparatively high performance compared to others. We built the abstract syntax tree (AST) of each source file using JDT and traversed the AST to extract four different document fields such as class names, method names, variable names, and comments. Then we tokenized all the identifier names and comment words and used Indri toolkit [33] for indexing. We performed a separate search for each of document fields with queries and then summed document scores across all searches. For $Dataset_{NLS}$ and $Dataset_{PES}$, we found that BRTracer and BLIA have good performance. They use the same method [41] to analyze stack traces information. The formula as follows:

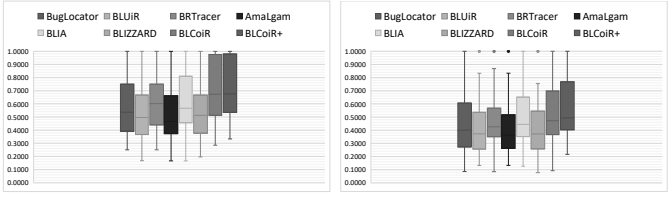
TABLE X: Comparing with State-of-the-Art Techniques.

Dataset	Technique	TOP1	TOP5	TOP10	MRR	MAP
$Dataset_{NL}$	BugLocator	18.07%	45.78%	55.42%	0.2938	0.2236
	BLUiR	27.71%	**51.81%***	**66.27%***	0.3858	0.2962
	BRTracer	25.30%	48.19%	63.86%	0.3623	0.2587
	AmaLgam	25.30%	50.60%	66.27%	0.3619	0.2821
	BLIA	15.66%	46.99%	62.65%	0.3023	0.2211
	BLIZZARD	18.07%	44.58%	54.22%	0.3080	0.2202
	BLCoiR	27.71%	46.99%	56.63%	0.3693	0.2668
	BLCoiR+	**31.33%***	49.40%	63.86%	**0.4103***	**0.3019***
$Dataset_{NLS}$	BugLocator	32.20%	57.63%	64.41%	0.4395	0.3143
	BLUiR	28.81%	54.24%	59.32%	0.4068	0.2809
	BRTracer	42.37%	64.41%	72.88%	0.5263	0.3670
	AmaLgam	30.51%	52.54%	55.93%	0.4044	0.2844
	BLIA	38.98%	59.32%	72.88%	0.4922	0.3700
	BLIZZARD	27.12%	47.46%	55.93%	0.3543	0.2662
	BLCoiR	30.51%	49.15%	57.63%	0.3816	0.2519
	BLCoiR+	**47.46%**	**71.19%**	**76.27%**	**0.5739**	**0.4124**
$Dataset_{PE}$	BugLocator	48.88%	68.73%	74.94%	0.5773	0.4236
	BLUiR	42.68%	64.02%	71.22%	0.5189	0.3848
	BRTracer	53.85%	72.21%	77.17%	0.6175	0.4548
	AmaLgam	43.18%	65.01%	71.71%	0.5244	0.3895
	BLIA	45.41%	70.47%	77.17%	0.5586	0.4290
	BLIZZARD	43.67%	68.49%	73.95%	0.5367	0.3900
	BLCoiR	**57.32%**	**82.13%**	**86.60%**	**0.6788**	**0.5109**
$Dataset_{PES}$	BugLocator	38.10%	65.99%	73.47%	0.5019	0.3776
	BLUiR	31.29%	53.74%	61.90%	0.4034	0.3083
	BRTracer	39.46%	76.19%	82.99%	0.5465	0.4175
	AmaLgam	31.29%	55.78%	63.95%	0.4131	0.3228
	BLIA	48.98%	74.15%	83.67%	0.6119	0.4949
	BLIZZARD	36.73%	53.74%	62.59%	0.4379	0.3487
	BLCoiR	45.58%	74.15%	79.59%	0.5729	0.4439
	BLCoiR+	**52.38%**	**81.63%**	**85.03%***	**0.6404**	**0.5028**
$Dataset_{All}$	BugLocator	38.95%	62.84%	70.89%	0.4961	0.3832
	BLUiR	34.08%	56.50%	64.58%	0.4396	0.3344
	BRTracer	44.12%	68.33%	75.66%	0.5460	0.4185
	AmaLgam	33.60%	55.79%	63.62%	0.4312	0.3305
	BLIA	40.67%	65.23%	75.47%	0.5216	0.4231
	BLIZZARD	34.24%	56.74%	63.87%	0.4358	0.3383
	BLCoiR	44.48%	63.49%	69.02%	0.5267	0.3797
	BLCoiR+	**49.15%***	**76.11%**	**81.74%**	**0.6045**	**0.4759**

$$BoostScore(x) = \begin{cases} \frac{1}{rank} & x \in D \& rank \leq 10 \\ 0.1 & x \in D \& rank > 10 \\ 0.1 & x \in C \\ 0 & otherwise \end{cases}$$

Here, rank stands for the position of the source file in stack traces, D is the set of files in the stack traces and C is the set of imported files of D.

We combined source code indexing and search method from BLUiR and stack traces analysis method from BRTracer and BLIA with our approach, and called it BLCoiR+.



(a) MRR. (b) MAP.

Fig. 6: Comparison of MRR and MAP with state-of-the-art techniques across projects.

As results shown in Table X, on average, BLCoiR+ significantly outperforms the other techniques. Especially, BLCoiR+ provides 11% higher MRR and 14% higher MAP than BRTracer, and 16% higher MRR and 12% higher MAP than BLIA. BLCoiR+ has better scores on five metrics on $Dataset_{NLS}$ and $Dataset_{PES}$. Additionally, BLCoiR+ outperforms BRTracer and BLIA on $Dataset_{NLS}$ and $Dataset_{PES}$. BLCoiR+ achieves the best MRR and MAP values on $Dataset_{NL}$, but they are not statistically distinguishable. None of the eight techniques substantially outperforms the others, all presenting a tight range of MAP and MRR values on $Dataset_{NL}$. This encourages improving IRFL for poor bug reports in future works.

We measured the performance (MAP and MRR) by applying each IRFL technique to each subject. Figure 6 shows the overall distributions of MAP and MRR for each technique. MRR and MAP values of all techniques are ranged from 0 to 1. Since some projects only have a few concurrency bug reports (e.g., CODEC, COLLECTIONS, COMPRESS, etc.), all techniques have the best top value reach 1 on MRR. The techniques BLUiR, BRTracer, AmaLgam, and BLIZZARD have higher outliers on MAP. The results indicate that our approach BLCoiR could achieve an equivalent performance with state-of-the-art techniques, and BLCoiR+ has the best performance compared to other techniques.

RQ3 summary: BLCoiR performed better than the other techniques on the $Dataset_{PE}$ and outperformed all the other techniques on average on five different evaluation metrics. However, BLCoiR did not perform as well as other techniques that utilized additional information such as stack trace information and source file analysis. Therefore, we combined their approach with those techniques and called it BLCoiR+, which outperformed all the other techniques on average on all five evaluation metrics. On the $Dataset_{NL}$, all the techniques, including BLCoiR and BLCoiR+, suffered from low performance. We suggest improving IRFL for poor bug reports in future works.

V. Threats to Validity

The primary threat to external validity for this study involves the representativeness of our datasets. Other datasets may exhibit different behaviors. However, we reduce this threat to some extent by using the widely adopted datasets from the state-of-the-art IRFL for our study. With these datasets, it provides a fair evaluation between BLCoiR and the the state-of-the-art. We cannot claim that our results can be generalized to all systems of all domains though. To overcome the generalizability, we consider mostly structured items (e.g., stack traces, program entities) from a bug report. Our approach can also be adopted to bug reports for other programming languages. To overcome the threat involving the generalizability of our extracted concepts and relations, we randomly selected 338 concurrency bug reports from our dataset, which contain

1,339 concurrency related sentences, to extract concepts and relations. We split each set randomly into ten equal subsets and performed iterations. After seven iterations, we observed that concepts and relations had become saturated, meaning that no new concepts or relations appeared. The primary threat to internal validity involves the use of keyword search and manual inspection to identify the subject bug reports (Section IV-A). However, combining keyword search and manual inspection is an effective technique to identify bugs of a specific type from a large pool of generic bugs and has been used successfully in prior studies [39], [48], [49].

VI. Related Work

IR-Based Fault Localization: IR-based fault localization is an effective technique for finding bugs in software systems. Automated techniques are desirable to avoid the time-consuming and error-prone manual inspection of code. Gay et al. [50] proposed a relevance feedback mechanism to improve the vector space model (VSM) used for retrieval. Zhou et al. [32] improved the VSM model by considering document length and proposed the revised VSM (rVSM). Saha et al. [40] used Eclipse JDT to parse the source code's abstract syntax tree (AST) and extract information from four types of code entities to improve IR-based fault localization. Wang et al. [42] proposed an approach to analyze version history information to enhance IR-based fault localization. Compared to existing techniques, our approach considers domain knowledge to formulate queries, which improves concurrency fault localization's performance.

Deep Learning-Based Approaches: Researchers have explored using deep learning techniques to enhance the accuracy of IR-based fault localization, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Lam et al. [51] proposed a novel approach that combines deep neural networks with rVSM. Xiao et al. [52] proposed a model that consists of a character-level CNN and an RNN language model. Xiao et al. [53] further improved on their previous work by proposing an enhanced CNN that considers bug-fixing recency and frequency. However, the size of the dataset is a limitation for these approaches since concurrency bugs are relatively rare, leading to poor performance. Future research should explore methods for addressing this issue, such as data augmentation techniques or transfer learning.

Graph-Based Approaches: Graph-based fault localization approaches have gained popularity due to their ability to identify the most important nodes in a program's dependency graph and locate the likely locations of bugs. PageRank is one commonly used approach. Scanniello et al. [54] proposed an approach that uses the PageRank algorithm to extract the dependencies of methods and facilitate a VSM-based fault localization model. Rahman et al. [55] further improved the technique by building a source token graph and using graph-based term weighting to reformulate the query. Rahman et al. [19] introduced BLIZZARD, which builds trace graphs, text graphs, and source token graphs, and uses PageRank to identify the most important nodes and reformulate the query. Our approach takes into account semantic dependencies and domain knowledge to improve fault localization's performance on concurrency bugs and outperforms BLIZZARD on concurrency fault localization.

VII. Conclusion

While IRFL has seen development, none of its prior applications targeted concurrent programs, despite their pervasive presence in modern software. In this paper, we propose a novel IRFL technique for concurrent programs, named BLCoiR, based on a knowledge graph (KG) representing domain entities from bug reports and their intricate relations. We evaluate BLCoiR on 692 concurrency bug reports from 44 applications, rigorously comparing it with six state-of-the-art IRFL techniques. Results affirm that BLCoiR excels, showcasing superior performance while exclusively considering bug report text. Further enriching our findings, we integrated techniques to analyze supplementary data, resulting in BLCoiR+ prevailing over alternatives. Our roadmap envisions the extension of our approach to diverse domain-specific programs, propelling its impact, and delving into finer-grained fault localization at the method level.

Acknowledgments

This work was supported in part by NSF awards CCF-2152340 and CCF-2140524.

References

[1] N. Shahmehri, M. Kamkar, and P. Fritzson, "Semi-automatic bug localization in software maintenance," in *Software Maintenance, 1990, Proceedings., Conference on*. IEEE, 1990, pp. 30–36.

[2] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[3] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 273–282.

[4] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proceedings of the International Conference on Software Engineering*, 2011, pp. 746–755.

[5] S. Roychowdhury and S. Khurshid, "Software fault localization using feature selection," in *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, 2011, pp. 11–18.

[6] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *ICSE*, 2010, pp. 245–254.

[7] R. Tzoref, S. Ur, and E. Yom-Tov, "Instrumenting where it hurts: an automatic concurrent debugging technique," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 27–38.

[8] S. Park, R. Vuduc, and M. J. Harrold, "A unified approach for localizing non-deadlock concurrency bugs," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, 2012, pp. 51–60.

[9] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps, "Conseq: detecting concurrency bugs through sequential errors," in *ACM SIGPLAN Notices*, vol. 46, no. 3, 2011, pp. 251–264.

[10] W. Wang, Z. Wang, C. Wu, P.-C. Yew, X. Shen, X. Yuan, J. Li, X. Feng, and Y. Guan, "Localization of concurrency bugs using shared memory access pairs," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 611–622.

[11] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 201–210.

[12] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *IEEE 31st International Conference on Software Engineering*, 2009, pp. 34–44.

[13] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*. IEEE, 2007, pp. 89–98.

[14] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, 2009, pp. 56–66.

[15] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 43–52.

[16] M. M. Rahman and C. Roy, "Poster: improving bug localization with report quality dynamics and query reformulation," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 348–349.

[17] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 842–851.

[18] O. Chaparro, J. M. Florez, and A. Marcus, "Using bug descriptions to reformulate queries during text-retrieval-based bug localization," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2947–3007, 2019.

[19] M. M. Rahman and C. K. Roy, "Improving ir-based bug localization with context-aware query reformulation," in *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 621–632.

[20] M. Kim, Y. Kim, and E. Lee, "A novel automatic query expansion with word embedding for ir-based bug localization," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 2021, pp. 276–287.

[21] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 232–242.

[22] M. M. Rahman and C. K. Roy, "Quickar: Automatic query reformulation for concept location using crowdsourced knowledge," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 220–225.

[23] B. Sisman and A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 309–318.

[24] L. Dietz, A. Kotov, and E. Meij, "Utilizing knowledge graphs for text-centric information retrieval," in *The 41st international ACM SIGIR conference on research & development in information retrieval*, 2018, pp. 1387–1390.

[25] R. Blanco and C. Lioma, "Graph-based term weighting for information retrieval," *Information retrieval*, vol. 15, no. 1, pp. 54–92, 2012.

[26] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.

[27] Naming conventions. [Online]. Available: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html

[28] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[29] G. Salton, "Introduction to modern information retrieval," *McGraw-Hill*, 1983.

[30] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[31] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[32] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *2012 34th International conference on software engineering (ICSE)*. IEEE, 2012, pp. 14–24.

[33] Lemur project. [Online]. Available: https://www.lemurproject.org/indri/

[34] Apache lucene. [Online]. Available: https://lucene.apache.org/

[35] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 689–699.

[36] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon, "Bench4bl: reproducibility study on the performance of ir-based bug localization," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 61–72.

[37] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 329–339.

[38] S. A. Asadollah, H. Hansson, D. Sundmark, and S. Eldh, "Towards classification of concurrency bugs based on observable properties," in *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*. IEEE, 2015, pp. 41–47.

[39] S. Abbaspour Asadollah, D. Sundmark, S. Eldh, and H. Hansson, "Concurrency bugs in open source software: a case study," *Journal of Internet Services and Applications*, vol. 8, no. 1, pp. 1–15, 2017.

[40] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 345–355.

[41] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *2014 IEEE international conference on software maintenance and evolution*. IEEE, 2014, pp. 181–190.

[42] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 53–63.

[43] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Information and Software Technology*, vol. 82, pp. 177–192, 2017.

[44] Openai. [Online]. Available: https://openai.com/

[45] E. M. Voorhees, "The TREC-8 question answering track report," in *Proceedings of The Eighth Text REtrieval Conference, TREC 1999, Gaithersburg, Maryland, USA, November 17-19, 1999*, ser. NIST Special Publication, E. M. Voorhees and D. K. Harman, Eds., vol. 500-246. National Institute of Standards and Technology (NIST), 1999.

[46] C. D. Manning, *Introduction to information retrieval*. Syngress Publishing,, 2008.

[47] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[48] F. Padberg, P. Pfaffe, and M. Blersch, "On mining concurrency defect-related reports from bug repositories."

[49] S. Abbaspour Asadollah, D. Sundmark, S. Eldh, H. Hansson, and E. P. Enoiu, "A study of concurrency bugs in an open source software," in *Open Source Systems: Integrating Communities: 12th IFIP WG 2.13 International Conference, OSS 2016, Gothenburg, Sweden, May 30-June 2, 2016, Proceedings 12*. Springer, 2016, pp. 16–31.

[50] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in ir-based concept location," in *2009 IEEE international conference on software maintenance*. IEEE, 2009, pp. 351–360.

[51] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 218–229.

[52] Y. Xiao and J. Keung, "Improving bug localization with character-level convolutional neural network and recurrent neural network," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 703–704.

[53] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Improving bug localization with word embedding and enhanced convolutional neural networks," *Information and Software Technology*, vol. 105, pp. 17–29, 2019.

[54] G. Scanniello, A. Marcus, and D. Pascale, "Link analysis algorithms for static concept location: an empirical assessment," *Empirical Software Engineering*, vol. 20, pp. 1666–1720, 2015.

[55] M. M. Rahman and C. K. Roy, "Improved query reformulation for concept location using coderank and document structures," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 428–439.