

Large-Scale Evaluation of Method-Level Bug Localization with FinerBench4BL

Shizuka Tsumita
Tokyo Institute of Technology
Tokyo 152–8550, Japan
tsumita@se.c.titech.ac.jp

Shinpei Hayashi
Tokyo Institute of Technology
Tokyo 152–8550, Japan
hayashi@c.titech.ac.jp

Sousuke Amasaki
Okayama Prefectural University
Okayama 700–0961, Japan
amasaki@cse.oka-pu.ac.jp

Abstract—Bug localization is an important aspect of software maintenance because it can locate modules that need to be changed to fix a specific bug. Although method-level bug localization is helpful for developers, there are only a few tools and techniques for this task; moreover, there is no large-scale framework for their evaluation. In this paper, we present FinerBench4BL, an evaluation framework for method-level information retrieval-based bug localization techniques, and a comparative study using this framework. This framework was semi-automatically constructed from Bench4BL, a file-level bug localization evaluation framework, using a repository transformation approach. We converted the original file-level version repositories provided by Bench4BL into method-level repositories by repository transformation. Method-level data components such as oracle methods can also be automatically derived by applying the oracle generation approach via bug-commit linking in Bench4BL to the generated method repositories. Furthermore, we tailored existing file-level bug localization technique implementations at the method level. We created a framework for method-level evaluation by merging the generated dataset and implementations. The comparison results show that the method-level techniques decreased accuracy whereas improved debugging efficiency compared to file-level techniques.

Index Terms—bug localization, information retrieval, repository transformation

I. INTRODUCTION

Bug localization is the process of identifying the location of a bug. Developers must fix many bugs in large-scale software projects, and debugging software is difficult and time-consuming [1]. As this can be a tedious task in large-scale software development projects, numerous ideas have been proposed to automate this process using software development information. For instance, we can identify the locations of a bug using the description of bug reports, *i.e.*, information retrieval (IR)-based techniques [2], [3], or execution traces, *i.e.*, dynamic analysis [4]. Several hybrid techniques that combine a base technique with additional information have been proposed to improve bug localization accuracy. For example, BugLocator [5] improved an IR-based technique with similar bug reports that were previously resolved. BLUIR [6] incorporated structural information in addition to using similar bug reports. AmaLgam [7] combined the version history, structural information, and similar bug reports.

Most existing IR-based bug localization (IRBL) techniques follow file-level recommendations. They output a ranked list of suspicious files; therefore, their evaluations were performed at

the file level. This granularity may be too coarse for developers to debug. In particular, IRBL approaches may recommend large files with more than 500 lines of code. Debugging efforts can be decreased if it is possible to recommend code fragments to be fixed at the method level.

However, in the literature, there are only a few IRBL techniques and their implementations at the method level [8]–[10]. To the best of our knowledge, no study has performed a comprehensive comparison and evaluation of method-level IRBL techniques, and only IR techniques have been compared [11]. In addition, no publicly available evaluation framework enables comparison at the method level, and we have little knowledge of method-level IRBL approaches.

Therefore, in this study, we propose FinerBench4BL, an evaluation framework for method-level bug localization techniques. This framework is based on Bench4BL [12], an existing evaluation framework for file-level bug localization techniques. We built a method-level bug localization dataset by applying a repository transformation to the repositories in the Bench4BL dataset and prepared method-level IRBL implementations by modifying the file-level ones.

This study also presents a performance study of method-level IRBL techniques. The results demonstrated that method-level IRBL techniques do not improve the accuracy but reduce debugging effort compared with file-level bug localization. We demonstrated a similar performance across different levels and revealed the need for improvements in method-level bug localization.

The main contributions of this study with respect to method-level bug localization techniques are as follows:

- a transformational approach to obtain method-level IRBL techniques and datasets,
- construction of an evaluation framework that enables comparison of IRBL techniques at the method level, and
- confirmation of the superiority and inferiority between techniques at the method level, which replicates existing IRBL approaches applied to the different levels.

The remainder of this paper is organized as follows. In the next section, we briefly introduce IR-based bug localization and its evaluation framework. Section III discusses the issues in terms of module granularity, and related work is introduced in Section IV. The approach proposed in this study that leads to the solution of these issues is presented in Section V.

TABLE I
APPLYING BUGLOCATOR TO CODEC-199

Rank	Filename	Score
1	Soundex.java	0.800
2	DaitchMokotoffSoundex.java	0.618
3	RefinedSoundex.java	0.564
4	SoundexTest.java	0.500
5	RefinedSoundexTest.java	0.388

Section VI presents the experimental setup used in the analysis and comparison of the results of the file and method levels. Threats to validity are presented in Section VII. Finally, the conclusions and future work are presented in Section VIII.

II. PRELIMINARIES

A. IR-Based Bug Localization

Bug localization is the process of finding buggy locations in source code based on the information about a given bug. This study explicitly focuses on IR-based bug localization (IRBL), which uses textual analysis to determine bug locations. IRBL techniques use a bug report and source code as inputs and output a ranked list of modules to be fixed. Some IRBL techniques may use past bug reports, stack traces, or historical information as additional inputs. The given bug report serves as a query for searching the given source code. A bug report includes the bug ID, summary, dates when the bug is opened or closed, and detailed description. The description of a bug may include stack traces that are extracted and utilized using advanced IRBL techniques. Finally, IRBL techniques compute the similarity score between the bug report and source code files considering the additional information and output a ranked list of source files on the score.

As an IRBL example, an excerpt result of applying BugLocator [5] to bug CODEC-199¹ is shown in Table I. The columns indicate the rank, filename, and similarity score of the target file for the query bug report. The highlighted row, `Soundex.java`, specifies the *oracle*, i.e., the buggy file for this bug. The higher the rank of the file, the more likely it is to contain a bug. Therefore, developers search for bugs starting with `Soundex.java` in the table.

B. IRBL Evaluation Framework

Researchers often employ a retrospective approach by applying IRBL techniques to previously resolved bugs to evaluate their performance. When resolving a bug, the list of fixed modules is regarded as the oracle list of modules that should be localized associated with the bug. A set of resolved bug reports and their associated lists of fixed modules form an IRBL evaluation dataset. The performance of IRBL techniques can be evaluated by comparing the oracle list of modules to the ranked list of modules produced by the techniques.

Bench4BL² [12] is a large-scale framework proposed by Lee *et al.* to evaluate the performance of file-level bug localization

techniques. Resolved bug reports obtained from the issue tracking systems for 46 projects and their lists of oracle source files to be localized were collected and provided as a dataset for evaluation. In addition to the dataset, they have attached the implementations of BugLocator [5], BLUIR [6], BRTracer [4], AmaLgam [7], BLIA [13], and Locus [14].

In Bench4BL, released versions of projects are regarded as source code snapshots to be searched. Each version was created by checking out files from a Git repository in the dataset. An oracle list of files corresponding to a bug report was generated based on the commit history in a repository. Once a bug-fixing commit is detected based on the matching between its commit message and the ID of the bug, the changed files in the commit are regarded as files to be fixed to resolve the bug. In addition, AmaLgam and BLIA utilize the historical information (also obtained from the Git repository) to improve the bug localization accuracy. In summary, all information used in applying IRBL techniques follows the original Git repositories.

III. MOTIVATION

Most existing IR-based bug localization techniques localize buggy code at the file level. Techniques such as BugLocator [5], BLUIR [6], AmaLgam [7], and BLIA [13] all recommend suspicious modules at the file level. This section presents the challenges of file-level bug localization and method-level bug localization.

A. Challenges of File-Level Bug Localization

When recommending buggy files, IR-based bug localization techniques may produce very large files that contain methods unrelated to the bug, making it challenging to identify bug locations. For example, consider the bug CODEC-221³. For this bug, the three methods in `HmacUtils.java` must be fixed. This file is 729 lines long and includes 40 methods. Therefore, significant time is required to identify the bug location in the entire file. It would be more helpful if the three buggy methods were recommended to be fixed directly.

Table II lists the results of applying BLIA to this bug report at both file and method levels. This table shows the names of the modules recommended by BLIA with their ranks at both the file and method levels. The score represents the points used to rank each file, and LOC shows the lines of code of the module. Rows with the correct answers are highlighted. In this example, the correct answer at file level, `HmacUtils.java`, is recommended at the first rank with a score of 0.640. However, identifying the methods that need to be fixed in this file, which consists of more than 700 lines, is challenging. Conversely, all buggy methods are localized up to the fourth in the list at the method level, and the sum of their LOC is only 82. The use of method-level IRBL reduces the effort required to read the 729 lines of `HmacUtils.java` by 11%.

In addition, the ratio of buggy methods to the total number of methods in a buggy file is generally small. In the context

¹<https://issues.apache.org/jira/browse/CODEC-199>

²<https://github.com/exatoo/Bench4BL>

³<https://issues.apache.org/jira/browse/CODEC-221>

TABLE II
APPLYING BLIA AT FILE AND METHOD LEVELS TO CODEC-221

Rank	File level			Method level		
	Module	Score	LOC	Module	Score	LOC
1	HmacUtils.java	0.640	729	BaseNCodecInputStream#reset()	0.640	15
2	DigestUtils.java	0.251	752	HmacUtils#updateHmac(Mac,InputStream)	0.573	29
3	HmacUtilsTest.java	0.095	237	HmacUtils#updateHmac(Mac,byte[])	0.565	19
4	BaseNCodecInputStream.java	0.048	184	HmacUtils#updateHmac(Mac,String)	0.565	19

TABLE III
COMPARISON OF IRBL STUDIES

	Additional information	Method level	# techniques	# projects	Multi-version
Lee <i>et al.</i> (Bench4BL) [12]	✓		6	46	✓
Youm <i>et al.</i> (BLIA1.5) [10]	✓	✓	1	3	
Amasaki <i>et al.</i> [15]	✓	✓	1	43	✓
Razzaq <i>et al.</i> (BoostNSift) [8]	✓	✓	4	4	
Chakkrit <i>et al.</i> [11]		✓	4	2	
Our Approach	✓	✓	5	37	✓

of bug prediction, Hata *et al.* investigated the ratio of buggy methods in a target project to ascertain the effectiveness of fine-grained bug predictions [16]. The results showed that the median number of buggy methods was 1–2, whereas that of all methods was 8–22. Therefore, file-level recommendations may be inefficient in identifying bug locations.

B. Challenges of Method-Level Bug Localization

Currently, there is a lack of knowledge regarding method-level IRBL because only a few method-level techniques exist. To the best of our knowledge, only a few bug localization techniques, such as BLIA1.5 [10], FineLocator [9], or BoostNSift [8], work at the method level. Furthermore, there is no unified framework for evaluating techniques at the method level, and the performance differences between the granularity levels and techniques remain unclear. Therefore, it is necessary to establish an evaluation framework for various techniques.

IV. RELATED WORK

A. IRBL Approaches

To date, many techniques have been studied that recommend bug locations based on information retrieval. Some of these techniques add other information to the similarity between the source code and bug reports. BugLocator [5] uses similarity to previous bug reports and file size. BLUIR [6] uses structural information of source code. BRTracer [4] uses the stack trace information in bug reports. AmaLgam [7] and Locus [14] use historical data. BLIA [13] combines all of these to localize bugs. The aforementioned techniques localize bugs at the file level and were evaluated with limited data, such as old versions of JDT or AspectJ. After that, Bench4BL, an evaluation framework proposed by Lee *et al.* [12], evaluated these six techniques for 46 projects at the file level. They suggested that IRBL techniques should be evaluated more accurately using multiple-version matching, which searches for a version in which a bug report is submitted.

Method-level IRBL is considered to aid developers in debugging more, and several method-level IRBL techniques have

been proposed. BLIA1.5 proposed by Youm *et al.* extends BLIA for method level bug localization [10]. Amasaki *et al.* [15] tailored BLUIR to localize bugs at the method level by using information outside the method. BoostNSift was proposed by Razzaq *et al.* [8] to filter source code based on bug report text.

Each method-level bug localization technique was evaluated using different datasets. Youm *et al.* used AspectJ, SWT, and ZXing, whereas Amasaki *et al.* used a part of the Bench4BL dataset. Razzaq *et al.* compared BoostNSift with BLUIR, BugLocator, and BLIA, and used the dataset used by Youm *et al.* plus Eclipse. Numerous method-level techniques were not applied to large projects in these evaluations, and the datasets were not uniform.

However, only IR techniques without additional information have been evaluated under a unified framework at the method level. Chakkrit *et al.* investigated the effectiveness of four IR techniques, VSM, LDA, LSI, and Entity Metric, at the method level [11]. They proposed a metric named top- k LOC, the percentage of bug reports for which at least one buggy file is found in the top-ranked files with a cumulative sum of k lines of code, to investigate the performance of the techniques. They found that the settings, such as how texts are pre-processed and which part of the texts in bug reports are used, significantly impact the performance of method-level IR techniques. In addition, settings with good performance produce good results at any granularity. They also stated that performance evaluation using LOC is necessary because debugging developers' efforts to find bugs may differ, even if they show similar accuracy.

Table III shows the relationship between this study and previous studies. In this table, the column of the additional information indicates whether the techniques consider other information. The column of the method level shows whether the approach is evaluated at the method level. The column of the multi-version shows whether the techniques localize bugs with multiple-version matching.

TABLE IV
COMPARISON OF IRBL DATASETS

Name	Granularity	# projects	# bugs
iBUGS [17]	File	3	390
MoreBugs [18]	File	2	902
BugLinks [19]	File	2	5,046
Bench4BL [12]	File	46	9,459
Bugzbook [20]	File	29	21,253
FDS [21]	File	11	4,429
MDS [21]	Method	14	360
FinerBench4BL	Method	37	3,344

B. IRBL Datasets

Bug localization techniques are evaluated by comparing the list of modules produced by applying the techniques to bug reports resolved previously with the oracle list of modules that have actually been fixed when bugs were resolved. Several sets of resolved bug reports and oracle lists of fixed modules were packed and proposed as a bug localization evaluation dataset. Table IV summarizes bug localization datasets.

iBUGS [17] is a dataset developed by Dallmeier *et al.* MoreBugs [18] proposed by Rao *et al.* is an extended dataset for projects adopted by iBUGS by additionally attaching version history information. BugLinks [19] is a dataset proposed by Sisman *et al.* that contains non-Java projects. iBUGS, MoreBugs, and BugLinks are relatively small datasets, comprising only 2–3 projects.

Bench4BL is a large-scale bug localization evaluation framework proposed by Lee *et al.* This dataset consisted of 46 Java projects and 9,459 bug reports. In addition, the Bench4BL framework bundles several IRBL technique implementations so that users can easily execute the techniques for projects in the provided dataset. Akbar *et al.* proposed BugzBook [20], which is another large-scale bug localization dataset that contains over 20,000 bug reports from 29 projects developed in Java, C/C++, and Python.

To the best of our knowledge, most publicly available bug localization datasets are at the file level. As mentioned above, several existing studies on method-level bug localization internally analyzed method-level datasets. However, these datasets are not publicly available. We believe that the inexistence of method-level datasets and evaluation frameworks hinders the promotion of method-level bug localization research. Note that Chappaoro *et al.* [21] prepared IRBL datasets at the class, file (FDS), and method level (MDS) as part of their study on query reformulation for bug localization. These datasets are available upon request.

V. APPROACH

To address the aforementioned challenges, we used the *method repositories* created by the repository transformation to construct a method-level IRBL dataset, and we modified the existing file-level IRBL implementations for the method-level ones. We constructed an evaluation framework for method-level IRBL techniques by combining them.

IRBL techniques output a list of source code files in the order of similarity to bug reports calculated using additional

information, such as the text of previous bug reports and historical data. They were evaluated by comparing the ranked list of modules obtained by applying IRBL techniques to resolve bug reports with the oracle list of modules. Accordingly, a method-level evaluation framework needs to fulfill the following three requirements:

- it should be able to output a ranked list of methods,
- it should be able to link bug reports to the fixed methods, and
- it should be able to obtain historical information about each method.

Most IRBL techniques assume that the input bug report and source code are in the form of files and calculate their similarity. They may consider the size or history of the source files as additional information to be used to calculate the score. A straightforward approach to make such implementations work at the method level is to add a specific process to extract information exclusively for the method of interest against existing IRBL implementations, which require substantial engineering work. Conversely, our idea is to prepare *method files* whose contents are only of the individual methods of interest, trick existing IRBL implementations to recognize that these method files are normal source files, and have them perform on these method files. This transforms the complexity of method-level bug localization into the cost of preparing the method files and minimizes the cost of re-implementing each IRBL implementation at the method level. Note that such method files are naturally considered incorrect Java source code for the project as a whole, but even such fake files can be analyzed without issues in most cases because IR-based approaches do not perform deep program analysis but only superficial text analysis. By preparing the method files using a *repository transformation* mechanism and converting every source file into method files at the stage of the original Git change history, most of the automated processing provided in the existing file-level IRBL evaluation framework, such as the generation of code snapshots from a repository, extraction of the change history of each file, and identification of oracle files based on the correspondence between bug reports and commits, could be completely reused. This could lead to an ecosystem of IRBL evaluation frameworks with multiple levels of granularity.

Accordingly, we propose an approach that transforms the repository itself, which records the change history at the method level, by applying a repository transformation mechanism. Figure 1 shows the repository before and after the transformation. As shown in the figure, the dataset repository provided by the existing evaluation framework was converted to generate source code files split by the method. Consequently, the entire dataset was converted to a method-level dataset. We then constructed a dataset that can compare IRBL techniques at the method level employing this method-level repository as the input. Fewer changes are required to modify IRBL techniques into method-level techniques. This approach is described in more detail as follows:

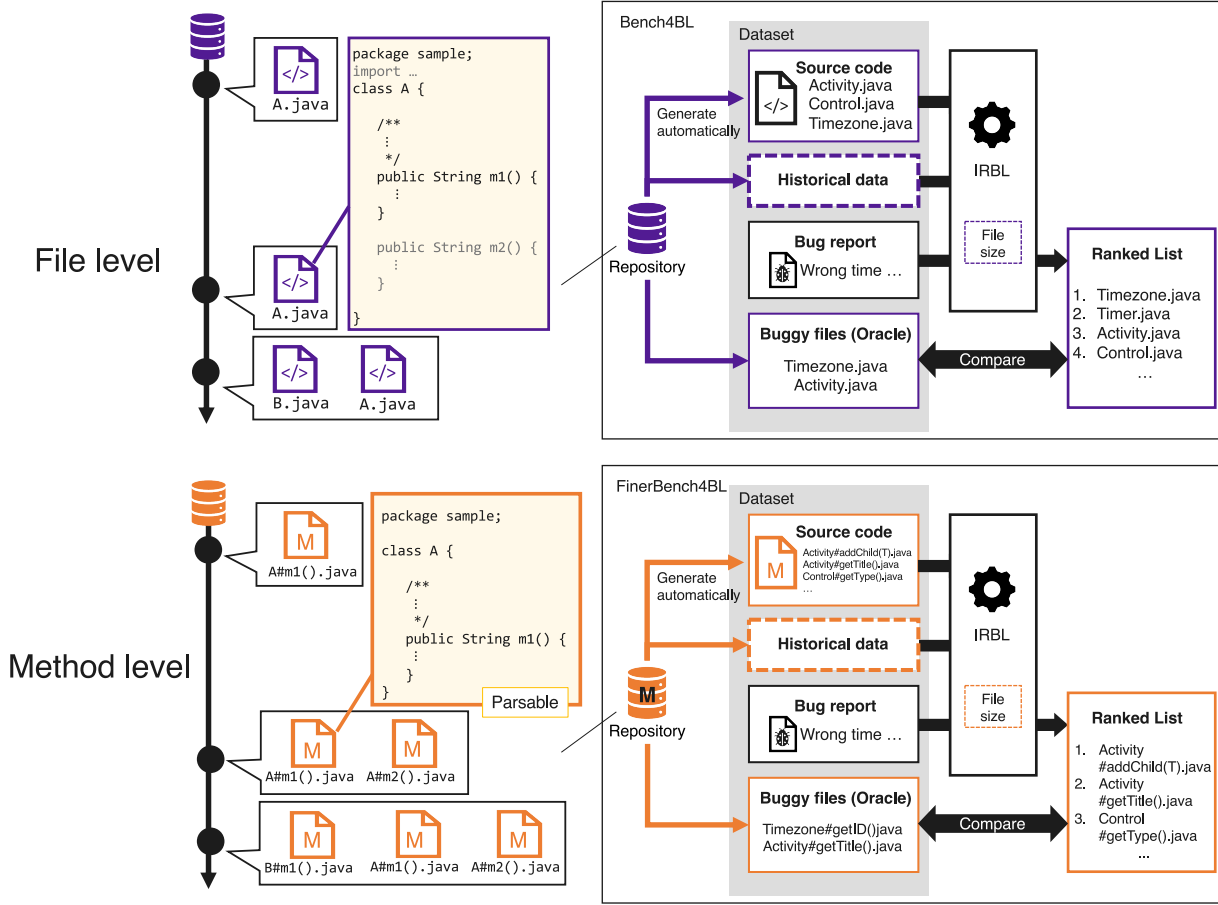


Fig. 1. Overview of FinerBench4BL.

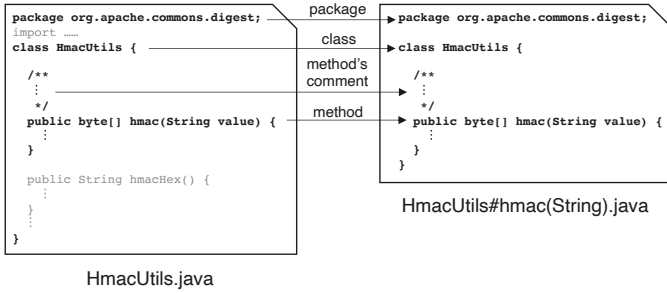


Fig. 2. Example of repository transformation.

A. Creating Method-Level Repositories

We used the Git repositories provided by Bench4BL and converted them into method-level repositories using Historinc [22], a repository transformation tool. We converted the target repository to the method level and obtained method files. Figure 2 demonstrates an example of the splitting part of file HmacUtils.java in the Apache Commons CODEC. By splitting the original source code files so that the results of parsing up to the method internals can be artificially reproduced. Therefore, if it merely uses the information available from a method file, such as file size or source code structure, no modifications are required.

body were extracted as the bodies of the method file. Therefore, in this approach, entities outside the method, such as fields and imports of the class, were excluded from the method file. Bugs caused outside of methods were excluded from the evaluation at the method level.

B. Generating Oracles

Bench4BL framework includes a script that links the resolved bug reports to fixed files. This script identifies the fixed file by linking bugs to commits that resolve them by referring to the bug ID in the commit message. Therefore, the link of bug reports can be updated by connecting bug reports to the method-level repository using the same script. This updated link can be used for the method file to compare method-level IRBL techniques in the Bench4BL framework.

C. Modifying IRBL Techniques to Method Level

We modified the existing IRBL techniques to output a ranked list at the method level. As explained, our approach splits source files into method files as parsable Java source code files so that the results of parsing up to the method internals can be artificially reproduced. Therefore, if it merely uses the information available from a method file, such as file size or source code structure, no modifications are required.

Furthermore, we could obtain historical information without modifying implementations because the repository itself is already fine-grained and consists of method-level contents. We need to modify an IRBL implementation only when it considers outside source code, such as the bug report description's stack trace, or when it requires valid source files.

For example, consider modifying BugLocator for method-level bug localization. BugLocator outputs the ranked list of files by considering the similarity between source code and a bug report, its file size, and the similarity to past bug reports. In this case, the original implementation could be completely reused without any changes because all the required information was available from the method file.

VI. EVALUATION

In this study, we compare and validate the method-level techniques modified by our approach and the method-level evaluation framework and answer the following research questions (RQs):

- RQ_1 : How much modification is required to convert IRBL techniques to the method level?
- RQ_2 : How well do the method-level IRBL techniques perform?

Five of the six techniques provided by Bench4BL were used for evaluation: BugLocator [5], BLUIR [6], BRTracer [4], AmaLgam [7], and BLIA [13], excluding Locus [14]. We excluded Locus because it crashed during execution, and we could not obtain the results. We found that several techniques provided by Bench4BL had issues that prevented them from correctly calculating the similarities. Therefore, we utilized BLIA implementation to imitate BLUIR, AmaLgam, and BRTracer by changing specific parameters to drop additional information to avoid the issues in their implementations.

A. RQ_1 : How much modification is required to convert IRBL techniques to the method level?

1) *Motivation*: We investigated the number of modifications to clarify whether our approach can easily convert the existing IRBL technique implementations to the method level.

2) *Study Design*: This study is based on the LOC of the modifications required to convert the techniques and the percentage of the total LOC of Java source files for each technique implementation.

3) *Results*: Figure V illustrates the results for RQ_1 . We needed to modify only BRTracer and BLIA using the proposed approach. These techniques for calculating similarity add scores to files included in the stack traces in the bug report. We modified the process of obtaining the file name from the stack trace to obtain the method names belonging to the file. For example, the stack trace of bug COMPRESS-203⁴ includes `org.apache.commons.compress.archivers.tar.TarArchiveOutputStream.writePaxHeaders(TarArchiveOutputStream.java:485)`. In this case, the modified method-level techniques were needed to obtain

TABLE V
CHANGES REQUIRED TO FIX TECHNIQUES FOR METHOD LEVEL

IRBL technique	Whole LOC	Modified LOC	Ratio (%)
BugLocator	3,180	0	0
BLUIR	10,469	0	0
BRTracer	10,530	4	0.038
AmaLgam	10,469	0	0
BLIA	10,474	4	0.038

TABLE VI
TARGET PROJECTS

Group	Project	# files	# methods	# versions	# bugs
Commons	CODEC	115	1,310	6	27
	COLLECTIONS	525	6,997	5	59
	COMPRESS	265	2,591	15	105
	CONFIGURATION	447	6,073	11	107
	CRYPTO	82	488	1	4
	CSV	29	452	3	6
	IO	227	2,608	12	70
	LANG	305	6,336	15	158
	MATH	1,617	15,695	15	175
	WEAVER	113	473	1	1
Jboss	ENTESB	252	3,210	1	4
	JBMETA	858	4,834	3	15
Spring	AMQP	408	3,996	32	86
	ANDROID	305	3,582	2	8
	BATCH	1,732	10,071	33	335
	BATCHADM	243	1,298	4	16
	DATACMNS	604	4,512	30	104
	DATAGRAPH	848	5,190	14	43
	DATAJPA	330	2,002	32	107
	DATAMONGO	622	6,703	40	209
	DATAREDIS	551	9,488	15	44
	DATAREST	414	2,183	23	89
	LDAP	566	3,556	5	46
	MOBILE	64	814	3	8
	ROO	1,109	7,803	15	568
	SEC	1,618	9,295	41	422
	SECOAUTH	726	3,912	6	61
	SGF	695	5,790	19	83
	SHDP	1,102	6,348	8	37
	SHL	151	749	2	6
	SOCIAL	212	1,344	4	10
	SOCIALFB	253	1,786	4	11
	SOCIALLI	180	830	1	2
	SOCIALTW	153	1,197	5	6
	SPR	6,512	57,696	10	89
	SWF	808	6,864	19	101
	SWS	925	3,505	24	122
Total		25,966	211,581	479	3,344

the method name `writePaxHeaders` and add the similarity score of the corresponding method file. It was easy to find the code location of the feature extracting the source file name from a stack trace and to modify it to extract the method file names by string manipulation, leading to only four line modifications.

Table V shows that the modifications of the two techniques were minimal (0.038 % each). Conversely, there were no modifications to techniques that use information directly related to files, such as file size and historical information.

Our approach could convert five existing bug localization techniques to method level in eight lines. The amount of modification was small, and it was easy to identify the

⁴<https://issues.apache.org/jira/browse/COMPRESS-203>

TABLE VII
RESULTS OF MAP

Project	File level					Method level				
	BugLocator	BLUIR	BRTracer	AmaLgam	BLIA	BugLocator	BLUIR	BRTracer	AmaLgam	BLIA
CODEC	0.631	0.623	0.283	0.629	0.621	0.192	0.352	0.094	0.345	0.381
COLLECTIONS	0.572	0.604	0.283	0.585	0.614	0.366	0.369	0.094	0.377	0.394
COMPRESS	0.631	0.703	0.263	0.678	0.696	0.281	0.318	0.170	0.302	0.349
CONFIGURATION	0.715	0.735	0.250	0.734	0.776	0.210	0.329	0.138	0.313	0.363
CRYPTO	0.314	0.313	0.063	0.322	0.323	0.106	0.060	0.264	0.060	0.058
CSV	0.806	0.833	0.482	0.833	0.833	0.155	0.423	0.154	0.437	0.453
IO	0.742	0.761	0.271	0.764	0.772	0.400	0.506	0.227	0.499	0.495
LANG	0.696	0.710	0.277	0.707	0.738	0.374	0.503	0.167	0.525	0.536
MATH	0.499	0.553	0.145	0.571	0.578	0.263	0.355	0.122	0.355	0.375
WEAVER	0.321	0.079	0.125	0.079	0.167	0.068	0.083	0.015	0.083	0.038
ENTESB	0.119	0.431	0.399	0.429	0.329	0.031	0.167	0.350	0.166	0.214
JBMETA	0.359	0.278	0.150	0.318	0.315	0.146	0.226	0.035	0.226	0.214
AMQP	0.404	0.421	0.086	0.422	0.434	0.155	0.167	0.106	0.165	0.187
ANDROID	0.540	0.562	0.566	0.499	0.556	0.298	0.343	0.237	0.353	0.367
BATCH	0.414	0.436	0.122	0.448	0.448	0.216	0.222	0.138	0.227	0.241
BATCHADM	0.438	0.553	0.276	0.549	0.606	0.256	0.223	0.136	0.248	0.225
DATA CMNS	0.300	0.365	0.136	0.369	0.362	0.125	0.204	0.124	0.201	0.214
DATAGRAPH	0.145	0.171	0.107	0.182	0.197	0.145	0.171	0.106	0.182	0.197
DATAJPA	0.355	0.369	0.110	0.380	0.394	0.169	0.173	0.115	0.175	0.183
DATAMONGO	0.296	0.317	0.111	0.316	0.340	0.114	0.144	0.096	0.142	0.165
DATAREDIS	0.382	0.410	0.114	0.406	0.391	0.163	0.160	0.123	0.152	0.190
DATAREST	0.264	0.272	0.119	0.290	0.289	0.100	0.127	0.077	0.128	0.128
LDAP	0.498	0.476	0.223	0.487	0.508	0.227	0.300	0.212	0.309	0.309
MOBILE	0.530	0.427	0.251	0.427	0.427	0.372	0.627	0.376	0.627	0.595
ROO	0.414	0.375	0.127	0.384	0.414	0.200	0.193	0.087	0.200	0.228
SEC	0.505	0.533	0.221	0.545	0.559	0.299	0.334	0.186	0.339	0.362
SECOAUTH	0.434	0.426	0.157	0.429	0.457	0.301	0.275	0.176	0.285	0.318
SGF	0.415	0.380	0.155	0.384	0.404	0.182	0.159	0.123	0.168	0.169
SHDP	0.387	0.361	0.187	0.360	0.379	0.200	0.161	0.166	0.160	0.211
SHL	0.503	0.593	0.217	0.593	0.593	0.227	0.327	0.226	0.327	0.314
SOCIAL	0.692	0.570	0.292	0.600	0.684	0.316	0.499	0.406	0.499	0.501
SOCIALFB	0.666	0.470	0.222	0.495	0.520	0.184	0.300	0.328	0.318	0.318
SOCIALLI	0.327	0.300	0.084	0.300	0.303	0.511	0.514	0.505	0.514	0.515
SOCIALTW	0.693	0.462	0.694	0.462	0.512	0.488	0.320	0.193	0.320	0.282
SPR	0.415	0.277	0.119	0.321	0.339	0.203	0.167	0.141	0.227	0.254
SWF	0.461	0.424	0.253	0.411	0.476	0.212	0.243	0.191	0.229	0.271
SWS	0.270	0.257	0.105	0.264	0.285	0.272	0.256	0.095	0.265	0.283

required changes.

B. RQ₂: How well do the method-level IRBL techniques perform?

1) *Motivation*: We evaluated modified techniques in a large-scale unified framework to identify changes in performance to gain new insights into method-level bug localization.

2) *Study Design*: We compared the performance of the techniques at the same and different levels. We evaluated the accuracy and effort required to determine the bug locations. MAP and MRR were used as measures of accuracy, and the top- k LOC [11] was used as a measure of effort. Top- k LOC is the percentage of bug reports for which at least one buggy file is found in the top-ranked files with a cumulative sum of LOC below k . For example, a top- k LOC of 0.1 for $k = 10,000$ means that the correct file appears within top 10,000 lines for 10% of bug reports. We used $k \in \{100, 500, 1000, 5000\}$. We used a two-sided Wilcoxon signed-rank test to evaluate the output of the techniques at both levels.

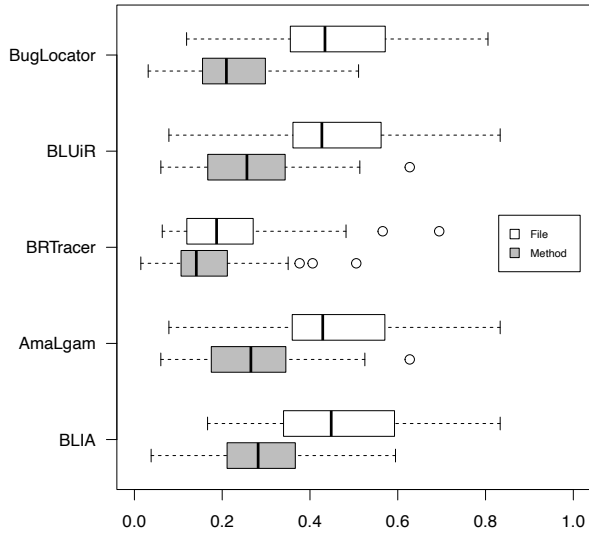
3) *Construction of FinerBench4BL Dataset*: In this experiment, we used 37 of the 46 projects provided by Bench4BL to

construct the FinerBench4BL dataset owing to the execution time. We filtered out bug reports that 1) any of the five IRBL techniques at either level failed to produce any solution. A typical example case to be filtered out is that the code location to be fixed when resolving the bug was outside of methods, and the list of solution modules at the method level became empty. The target projects, number of modules, versions, and bug reports to be used from Bench4BL and to be generated as FinerBench4BL are shown in Table VI. 25,966 original source files and 211,581 method files from all 479 versions were searched for 3,344 bug reports.

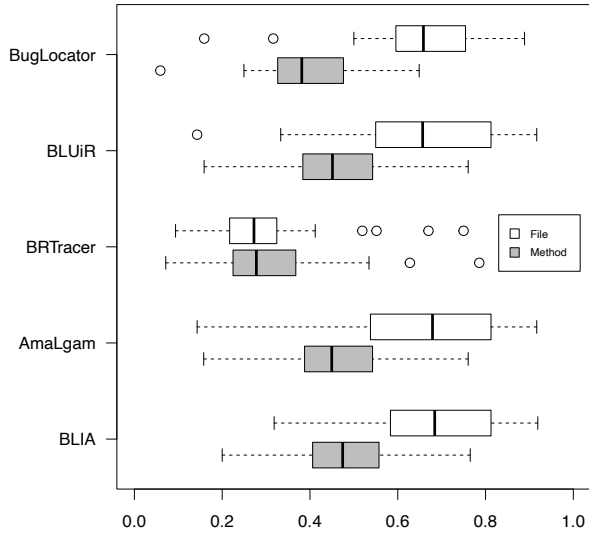
4) *Results of Accuracy Evaluation*: Table VII presents the MAP results for each project. Box plots of the MAP and MRR of the projects comparing the file and method levels are shown in Figs. 3(a) and 3(b). We describe only the MAP results because the MRR results showed a similar tendency to the MAP results. The technique with the highest accuracy for the same project is highlighted as bold in the results in Table VII.

First, we investigated the accuracy at each level. BLIA showed the highest accuracy, with the highest values in 20 of the 37 projects at the file level and 26 at the method level.

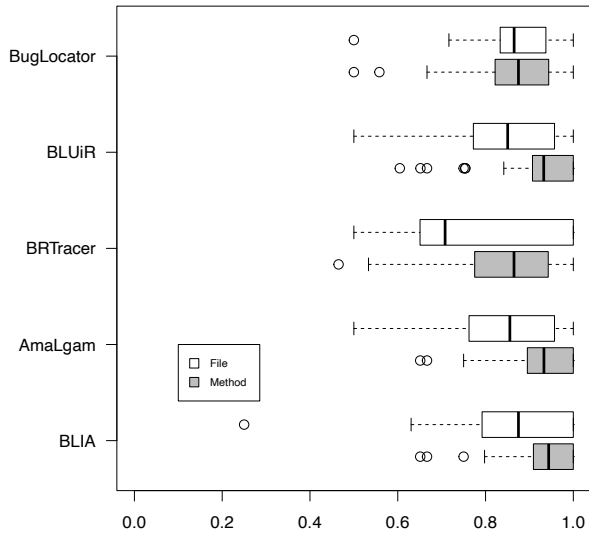
Furthermore, the same techniques showed the highest MAP



(a) MAP.



(b) MRR.



(c) Top-1,000 LOC.

Fig. 3. Evaluation of IRBL techniques at each level.

at both levels for 18 projects (48.6% of the total). This means that, for each project, the technique that showed the highest accuracy at the file level is likely to recommend buggy methods precisely at the method level. This result suggests that the selection of effective techniques for file-level bug localization may also be helpful at the method level.

Second, we investigated how the accuracy of each technique changed as it was modified for the method level. There were significant differences in the median MAP between the techniques. MAP at the file level was 1.33 times higher than that at the method level for BRTracer as the minimum and 2.07 times higher for BugLocator as the maximum. Furthermore, at both levels, the median MAP value indicated by BLIA was the largest, followed by AmaLgam, BLUIR, BugLocator, and BRTracer. Therefore, the accuracy increased with the amount of additional information handled, except for BRTracer. The high accuracy of BLIA, which uses most of the information, indicates that it can maintain accuracy by supplementing the missing information at method-level bug localization from various perspectives. In contrast, BLUIR, which uses only structural information as additional information, showed an accuracy comparable to that of AmaLgam and BLIA, which also consider other information. This suggests that structural information is adequate for method-level bug localization.

In some bugs, significant changes in accuracy occur with different bug localization granularities. Two examples of BugLocator outputs are presented below.

The first case is of decreased accuracy. In COLLECTIONS-220⁵, the `writeObject()` in `UnboundedFifoBuffer.java` was the cause of the bug. Although the target file was ranked first at the file level, `writeObject()` was recommended in 789th place at the method level. The words “increment”, “tail”, and “head” in this bug report were not included in any `writeObject()`, which was a deficient information method for the seven lines. However, `add()`, `remove()`, and other methods included in `UnboundedFifoBuffer.java`, contained several relevant words. This suggests that information from irrelevant methods contributes to the high accuracy in file-level bug localization.

The second case is of improved accuracy. In CONFIGURATION-558⁶, `getList()` in `MultiFileHierarchicalConfiguration.java` was the buggy method. The method-level bug localization improved the rank from 22nd to seventh at the file level. Although there were few bug report descriptions, they included the parameters and method names directly related to the target method. These entities might have contributed to the improvement in the accuracy at the method level. This is supported by the results of Wang *et al.* [23] and Rahman *et al.* [24], who showed that bug reports containing program entities are suitable for IRBL.

These examples with significant variations in accuracy at both levels suggest that, at the file level, information on methods irrelevant to the bug location is used for information

⁵<https://issues.apache.org/jira/browse/COLLECTIONS-220>

⁶<https://issues.apache.org/jira/browse/CONFIGURATION-558>

TABLE VIII
MEDIAN OF TOP-1,000 LOC VALUES

	File level	Method level	p-value	Cliff's d
BugLocator	0.865	0.875	0.644	0.063 (negligible)
BLUIR	0.850	0.933	0.026	0.297 (small)
BRTracer	0.708	0.865	0.024	0.301 (small)
Amalgam	0.855	0.933	0.022	0.305 (small)
BLIA	0.875	0.944	0.027	0.295 (small)

retrieval. This suggests the output ranked list based on unrelated methods may not help developers find such bug locations. Conversely, the textual information of the method body may be insufficient for information retrieval at the method level, which is considered to have reasonable granularity. Therefore, for method-level IRBL, it is useful to implement a hybrid approach for bug localization that uses not only the information on the target method body but also the information outside of its own method, as proposed by Amasaki *et al.* [15].

BLIA showed the highest accuracy at both levels. In 48.6% of the projects, the best-performed techniques were the same at both levels. Therefore, an accurate technique at the file level performs well at the method level. Furthermore, we found that the additional information effectively contributed to the recommendation of the buggy methods despite a 2.07-fold difference in accuracy.

5) *Results of Effort Evaluation*: Figure 3(c) shows the results of the top-1,000 LOC, which is a measure of effort to find bug locations from the ranked list. Table VIII also presents the top-1,000 LOC median, p-value of the two-sided Wilcoxon signed-rank test, and Cliff's d with an interpretation [25] for each project. Owing to space limitations, we omit the results using $k \in \{100, 500, 5000\}$ because they produce similar tendencies to the case of $k = 1000$.

For all techniques, top-1,000 LOC performance was improved at the method level. Among the four techniques, except for BugLocator, there were significant differences in the top-1,000 LOC between the two levels.

Compared to the top-1,000 LOC performance at the file level, BugLocator showed the smallest increase at 2.2%, and BRTracer showed the largest increase at 22.2%. At the method level, the best performing BLIA identified correct files within the top-1,000 lines of code of the ranked list for 94.4 % of the bug reports.

The top-1,000 LOC performance of the file level was lower than that of the method level owing to the large size of each file's lines of code, despite better accuracy. This suggests that the number of lines developers need to read could be reduced by current method-level IRBL approaches, even if their accuracy is not high.

In terms of the performance order, BLIA showed the highest top-1,000 LOC, and BRTracer showed the lowest, at both levels. The order of the method level performance of the top-1,000 LOC for each technique is the same from the file level, as with the results of the accuracy evaluation, except for

BugLocator, which did not show significant differences. This indicates that a technique with a good effort performance at the file level also performed well at the method level. Furthermore, as in the above discussion of accuracy, BLUIR, which only uses structural information, performed top-1,000 LOC as well as the other techniques at both levels. Therefore, the consideration of structural information leads to an improvement in the top-1,000 LOC performance.

Except for one technique, the method-level techniques significantly improved the top-1,000 LOC. BLIA, which best performed, found fixed method files in more than 94% of bug reports within the top 1,000 lines of code of the rank list. Techniques that perform well at the file level often also perform well at the method level. Structural information may improve the performance in method-level bug localization, where information is scarce.

VII. THREATS TO VALIDITY

In this study, we assumed that bug reports were correctly linked with fixed files. We used scripts provided by Bench4BL, linked bug reports, and fixed files. However, we did not check the validation of the correctness of the linking; therefore, there may be bug reports that cannot be identified in the commit log and those linked with non-buggy files.

Although the top- k LOC was adopted as an indicator for evaluating the effort, its validity to the actual effort required to identify bug locations was not apparent. In addition, we used the total number of LOC from the top of the ranked list above the correct solution file as an indicator of effort. However, if the solution is ranked at the top, then the total number is zero. This result may differ from the actual effort. Moreover, this indicator also affects the evaluation in terms of effort as developers do not necessarily read all lines of files.

VIII. CONCLUSION

In this study, we propose an approach to convert IRBL techniques to the method level using repository transformation. We evaluated them at both method and repository levels using a framework combining converted techniques with datasets constructed at the method level. This approach can convert the existing IRBL techniques to method-level techniques with minor modifications.

The evaluation results showed that the converted method-level techniques decreased the accuracy but reduced the debugging effort. However, as Amasaki *et al.* [15] stated, there is potential for further performance improvement using file-level information to compensate for the lack of details in method-level bug localization. In future research on method-level IRBL techniques, obtaining performance improvements will be easier using our framework to adjust the parameters and select additional information.

The future research directions of this study are as follows:

- *Increasing the number of evaluated techniques and datasets*. We believe the evaluation framework can be

extended to existing method-level techniques and other datasets to gain more knowledge.

- *Tuning the parameters of method-level techniques.* The method-level techniques in this paper are run with parameters tuned for the file level. Further performance improvements could be achieved by identifying optimal settings for method-level bug localization.
- *Comparing both levels of techniques under more uniform conditions.* When it is necessary to find all methods in a fixed file, the number of solutions that method-level techniques need to recommend is higher than at the file level, contributing to a significant reduction in accuracy. Bug reports with many solution methods should be excluded and compared under appropriate conditions.
- *Considering code elements outside of methods.* Incorporating information in Java source files, such as fields and imports, may provide different insights, which were excluded in this study.

The FinerBench4BL experimental results of this study are publicly available [26].

ACKNOWLEDGMENTS

This study was partially supported by JSPS KAKENHI (JP21H04877, JP21K18302, JP21KK0179, 21K11833, and JP22H03567).

REFERENCES

- [1] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2006.
- [2] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.
- [3] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, 2011, pp. 263–272.
- [4] 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 *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, 2014, pp. 181–190.
- [5] 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 *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012, pp. 14–24.
- [6] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*, 2013, pp. 345–355.
- [7] 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 (ICPC'14)*, 2014, pp. 53–63.
- [8] A. Razzaq, J. Buckley, J. V. Patten, M. Chochlov, and A. R. Sai, "Boost-NSift: A query boosting and code sifting technique for method level bug localization," in *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'21)*, 2021, pp. 81–91.
- [9] W. Zhang, Z. Li, Q. Wang, and J. Li, "FineLocator: A novel approach to method-level fine-grained bug localization by query expansion," *Information and Software Technology*, vol. 110, pp. 121–135, 2019.
- [10] 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.
- [11] C. Tantithamthavorn, S. Lemma Abebe, A. E. Hassan, A. Ihara, and K. Matsumoto, "The impact of IR-based classifier configuration on the performance and the effort of method-level bug localization," *Information and Software Technology*, vol. 102, pp. 160–174, 2018.
- [12] 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 (ISSTA'18)*, 2018, pp. 61–72.
- [13] K. C. Youm, J. Ahn, J. Kim, and E. Lee, "Bug localization based on code change histories and bug reports," in *Proceedings of the 22nd Asia-Pacific Software Engineering Conference (APSEC'15)*, 2015, pp. 190–197.
- [14] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*, 2016, pp. 262–273.
- [15] S. Amasaki, H. Aman, and T. Yokogawa, "On the effects of file-level information on method-level bug localization," in *Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'20)*, 2020, pp. 314–321.
- [16] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012, pp. 200–210.
- [17] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, 2007, pp. 433–436.
- [18] S. Rao and A. Kak, "moreBugs: A new dataset for benchmarking algorithms for information retrieval from software repositories," Purdue University, School of Electrical and Computer Engineering, Tech. Rep. TR-ECE-13-07, 2013. [Online]. Available: <https://engineering.purdue.edu/RVL/Database/moreBugs/TechReport.pdf>
- [19] B. Sisman and A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*, 2013, pp. 309–318.
- [20] S. A. Akbar and A. C. Kak, "A large-scale comparative evaluation of IR-based tools for bug localization," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'20)*, 2020, pp. 21–31.
- [21] 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.
- [22] S. Shiba and S. Hayashi, "Historinc: A repository transformation tool for fine-grained history tracking," *Computer Software*, vol. 39, no. 4, pp. 75–85, 2022.
- [23] Q. Wang, C. Parnin, and A. Orso, "Evaluating the usefulness of IR-based fault localization techniques," in *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA'15)*, 2015, pp. 1–11.
- [24] M. M. Rahman and C. K. Roy, "Improving IR-based bug localization with context-aware query reformulation," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*, 2018, pp. 621–632.
- [25] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys?" in *Proceedings of the 2006 Annual Meeting of the Florida Association of Institutional Research (FAIR'06)*, 2006, pp. 1–33.
- [26] S. Tsumita, S. Hayashi, and S. Amasaki, "Appendix of large-scale evaluation of method-level bug localization with FinerBench4BL," Zenodo, 2023. [Online]. Available: <https://doi.org/10.5281/zenodo.7546043>