CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Data Mining for Defects in Multicore Applications: An Entropy-Based Call-Graph Technique

Frank Eichinger[1], Victor Pankratius[2*] and Klemens Böhm[3]

[1]*SAP Research Karlsruhe, Germany*
[2]*Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA*
[3]*Karlsruhe Institute of Technology (KIT), Germany*

SUMMARY

Multicore computers are ubiquitous. Expert developers as well as developers with little experience in parallelism are now asked to create multithreaded software in order to exploit parallelism in mainstream shared-memory hardware. However, finding and fixing parallel programming errors is a complex and arduous task. Programmers thus rely on tools such as race detectors that typically focus on reporting errors due to incorrect usage of synchronization constructs or due to missing synchronization. This arsenal of debugging techniques, however, is incomplete. This article presents a new perspective and addresses a largely unexplored direction of defect localization where a wrong usage of *non-parallel* programming constructs might cause wrong *parallel* application behavior. In particular, we make a contribution by showing how to use data-mining techniques to locate defects in multithreaded shared-memory programs. Our technique analyzes execution anomalies in a condensed representation of the dynamic call graphs of a multithreaded object-oriented application and identifies methods that contain a defect. Compared to race detectors that concentrate on finding incorrect synchronization, our method is able to reveal a wider range of defects that affect the control flow of a parallel program. Results from controlled experiments show that our data-mining approach not only finds race conditions in different types of multicore applications, but also other errors that cause incorrect parallel program behavior. Data-mining techniques offer a fruitful new ground for parallel program debugging, and we also discuss long-term directions for this interesting field.

Received ...

KEY WORDS: parallel computing; software defect localization; multithreaded programs; dynamic call graphs; data mining

1. INTRODUCTION

Multicore computers with several cores on a chip provide developers with new opportunities to increase performance, but applications need to be multithreaded to exploit the hardware potential [1]. Compared to sequential software development, programmers are now additionally confronted with nondeterminism and parallel-programming errors, such as race conditions or deadlocks [2, 3]. A significant part of the developer community, however, has little experience with parallel programming. Even experts have difficulties finding parallel programming errors in complex applications. Effective debugging tools are needed to ensure good parallel application quality, in the following sense.

Static and dynamic debugging aids for parallel shared-memory programs are widely available [2, 4, 5, 6, 7, 8, 9, 10]. They focus on identifying atomicity violations, race conditions, or deadlocks

*Correspondence to: pankratius@csail.mit.edu. Work done while all authors have been at KIT, Karlsruhe, Germany.

due to wrong or inconsistent locking. These tools specialize on a particular class of parallel programming errors that are due to wrong usage of synchronization constructs in parallel programming languages. This article presents a new perspective showing that little attention has been paid to other causes (e.g., originating from *non-parallel* constructs) that might produce wrong *parallel* program behavior. As a consequence, quality assurance with existing tools is incomplete and has to be tackled with appropriate techniques. Throughout this article, we use a more precise terminology to characterize the colloquial *"bug"*. In particular, we distinguish between *defects*, *infections* and *failures*, according to Zeller [11]: *Defects* are the positions in the source code which cause a problem, an *infection* is an incorrect program state (usually triggered by a defect), and *failures* are an observable incorrect program behavior (e.g., a user obtains wrong results).

Let us consider some motivating examples. First, suppose that a programmer forgets or incorrectly specifies a condition when she or he writes the code creating threads in a thread pool. This slip affects parallel behavior and might lead to an unbounded creation of threads, wrong control flow and incorrect program outputs. As a second example, think of a programmer who incorrectly uses a sequential memory allocator in a multithreaded context in a language without automatic garbage collection. In rare cases, different threads could allocate overlapping parts of the memory and perform concurrent accesses, which leads to races. Even though race detectors would be able to intervene and show a report when a race occurs on a particular memory location, many tools offer little insight on the real cause of the problem. There is a clear need for more general defect-localization techniques to deal with such situations. Advances in this area are of great importance for industrial practice and the development of complex multithreaded programs.

This article addresses this new problem area and proposes a novel usage of data-mining techniques for defect localization in multithreaded shared-memory programs. Our approach is designed to detect a wider range of defects that affect parallel execution rather than just race conditions. In particular, we employ data mining on dynamic call graphs from a multithreaded object-oriented application to detect anomalies in program behavior. We compare the structure of the call graphs and the call frequencies from correct and incorrect program executions to isolate the methods that potentially contain defects. This procedure is motivated by our previous studies with single-threaded programs [12] where a similar approach has outperformed established techniques such as SOBER [13] and Tarantula [14, 15] with its extensions [16]. In this article we also discuss the effectiveness of different call-graph representations of multithreaded programs for call-graph mining. The article introduces a new graph representation with edge annotations that is robust in situations with varying thread schedules. This does away with the need for virtual machines that control thread schedules. Our approach thus requires less complex infrastructure support. In addition, our representation remains compact by summarizing redundant graph parts; this approach reduces overhead and improves the accuracy of the analysis. For instance, this holds in cases where multiple threads perform recursive calls, or in cases where similar work is done in parallel by replicated tasks. Contrary to race detectors that produce many warnings (most of which are false positives) in some arbitrary order, our technique determines a ranking of methods ordered by defect probability. Our controlled experiments with various types of applications show that call-graph mining finds defects in multithreaded programs. An upper bound of several hundred program executions suffices to pinpoint the defects. In addition, our approach has identified a previously unknown and undocumented defect in an open-source tool. The article extends our previous work on similar techniques for sequential software [17] and multithreaded programs [18] and presents additional details. We also discuss long-term directions of this fruitful field.

The article is organized as follows: Section 2 discusses related work. Section 3 explains the principles of call-graph-based defect localization and contrasts representations for multithreaded program call graphs. Section 4 introduces our approach to mine these graphs and use the results for defect localization. Section 5 shows a detailed example. Section 6 evaluates our approach and compares our technique with other approaches. Section 7 discusses long-term visions for research in data mining for defect localization in parallel programs. Section 8 provides a conclusion.

DOI: 10.1002/cpe

2. RELATED WORK

Defect identification techniques are typically classified into *static* and *dynamic* approaches. For both classes, we review previous work related to multithreaded programs as well as sequential programs.

2.1. Techniques for Defect Localization in Multithreaded Programs

Tools employing static analysis, such as RacerX by Engler and Ashcraft [4] or ESC/Java by Flanagan et al. [5], investigate the source code without execution. While they do not require program execution, a serious drawback is that they can produce a large number of false-positive warnings and may require significant manual code annotations to reduce this number.

Dynamic race detectors such as Eraser by Savage at al. [9] instrument programs and analyze runtime behavior of the memory access of each thread. Dynamic approaches can influence a program under test and change its timing, which can make a race condition disappear. This effect, known as the *probe effect* [19], needs to be avoided by effective debugging tools.

A problem of dynamic race detectors is that a race might manifest itself only when certain thread schedules occur. As scheduling is done by the operating system, developers have limited influence on reproducing a race. Addressing this problem, ConTest by Farchi et al. [2] executes a multithreaded Java program several times and influences thread schedules by inserting certain statements (e.g., `sleep()`) into a program. Chess, developed by Musuvathi et al. [6] for C#, has an additional refinement: a modified thread scheduler exhaustively tries out every possible thread interleaving. To reduce the search space, thread interrupts are only allowed at particular locations. On top of that, a *delta-debugging* strategy [20] might be used to automatically locate a defect. However, Tzoref et al. [10] have shown that approaches building on varying thread interleavings and delta debugging do not scale well for large software projects.

Hybrid race detectors such as the one by O'Callahan and Choi [7] and implementations such as the IBM MulticoreSDK by Qi et al. [8] combine different *dynamic* techniques to improve race detection. The MulticoreSDK also incorporates results from *static* analysis by identifying memory objects that can safely be excluded from further consideration. We compare results with our approach in Section 6.6.

In contrast to our approach, all of the tools mentioned in this section focus on finding synchronization errors due to wrong usage of parallel constructs. This is a subset of the errors that we can detect with our approach.

2.2. Techniques for Defect Localization in Sequential Programs

FindBugs [21] is a static code-analysis tool developed by Ayewah et al. It statically checks Java code for certain patterns of defect-prone artifacts. Although it supports a limited number of defect-prone multithreading-related behavior, it was originally not designed to detect multithreading defects. FindBugs complements our approach, as it can be used at an earlier stage of the development process (i.e., during coding rather than during testing). It has successfully been employed in a large-scale industrial setting [22]. However, FindBugs does not actually execute a program and often produces a large number of false positive warnings [23]. We compare its results with our approach in Section 6.6.

Dallmeier et al. [24] present a dynamic fault identification technique, but do not investigate multithreaded programs. In essence, it compares method sequence sets instead of statement coverage or call graphs. The authors demonstrate that the temporal order of calls is more promising to analyze than statement coverage only. More concretely, they compare object-specific sequences of incoming and outgoing object calls, using a sliding-window approach. Then they derive a ranking at the granularity level of classes, based on the information for which objects the statement sequences differ most between correct and failing executions. The extension of this technique for parallel programs is missing and non-trivial.

Tarantula by Jones et al. [14, 15] is a dynamic technique using tracing and visualization. To locate defects, it utilizes a ranking of basic blocks (sequences of statements without any branches in the control flow) which are executed more often in failing program executions. Though this technique

Prepared using cpeauth.cls
DOI: 10.1002/cpe

is rather simple, it produces accurate defect-location results in the single-threaded case. However, it does not take into account how often a statement is executed within one program execution, so certain defects might be missed. The tool was neither designed nor has it been evaluated with multithreaded programs. Spectrum-based fault localization techniques as employed in this tool are more generally described by Abreu et al. [16], though with a focus on sequential programs. A statistical approach similar to Tarantula was also implemented in [13]. All these approaches require significant improvements to work with parallel programs and thus are not directly comparable with the data-mining approach in our multithreaded scenarios.

3. USING DYNAMIC CALL GRAPHS AS A BASIS FOR DEFECT DATA MINING

This section explains how call graphs can provide a basis for defect data mining. Explorative studies with sequential applications have shown that there is great potential, but also the need for extensions of representations of program executions in parallel scenarios. After a discussion of trade-offs for various call-graph representations in parallel scenarios, we present our representation and show why it is beneficial in comparison to the potential alternatives. We also explain how our technique generates the call graphs that are used to mine our multithreaded applications for defects.

3.1. Explorative Studies of Graph-Mining in Sequential Programs

For sequential software, Liu et al. [25] and Di Fatta et al. [26] have proposed graph-mining techniques for defect localization, working on call graphs that represent program execution traces. The techniques assume that a collection of test cases is available, and that it is possible to decide if a program is executed correctly or not. Both approaches deal with *occasional bugs*, i.e., defects that lead to both correct and failing executions. As they do not consider multithreaded programs, this behavior depends on the input data, but it could be caused by varying thread interleavings, too. Furthermore, they focus on *non-crashing bugs*.

A detailed survey of call-graph-mining-based defect localization is presented in [27]. The core idea of most approaches is to mine for patterns in the call graphs that are characteristic for incorrect executions. Thereafter, they calculate its defect probability for each method. The call graphs may become huge, so it is necessary to work on a compact representation. In [17] we observe that the representations in [25, 26, 28] loose the information how many method calls an edge represents in the call graph. In [17], we therefore extend the graphs with edge weights representing call frequencies. We also demonstrate in [17] that data-mining analyses based on such graphs detect defects that other approaches cannot deal with. In [12], call-graph-mining-based techniques have outperformed established techniques we have discussed in the related work [13, 14, 15, 16] on the dataset from [17].

Other recent approaches introduce call graphs with several granularity levels, instead of one at the level of methods, such as the basic-block level used by Cheng et al. [28]. It facilitates more detailed defect localizations.

All these sequential techniques cannot be applied to multithreaded software right away. One reason is that they do not have call-graph representations working on multithreaded programs. Thus, two extensions are necessary: (1) Find an appropriate graph representation for multithreaded programs, and (2) adapt the mining scheme to work on the new graph representation. We address both issues in our article.

3.2. Dynamic Call Graphs for Multithreaded Programs

All call graphs are tracked during the runtime of an application. As there are several possibilities to represent call graphs for multithreaded programs, we discuss their advantages and disadvantages before we make a choice.

Unreduced call-graph representations contain the most detailed information on which thread calls which methods (see Figure 1(a)). Our approach requires call graphs at the granularity level of methods, i.e., nodes refer to methods and edges to method calls. In the multithreaded case, every

DOI: 10.1002/cpe

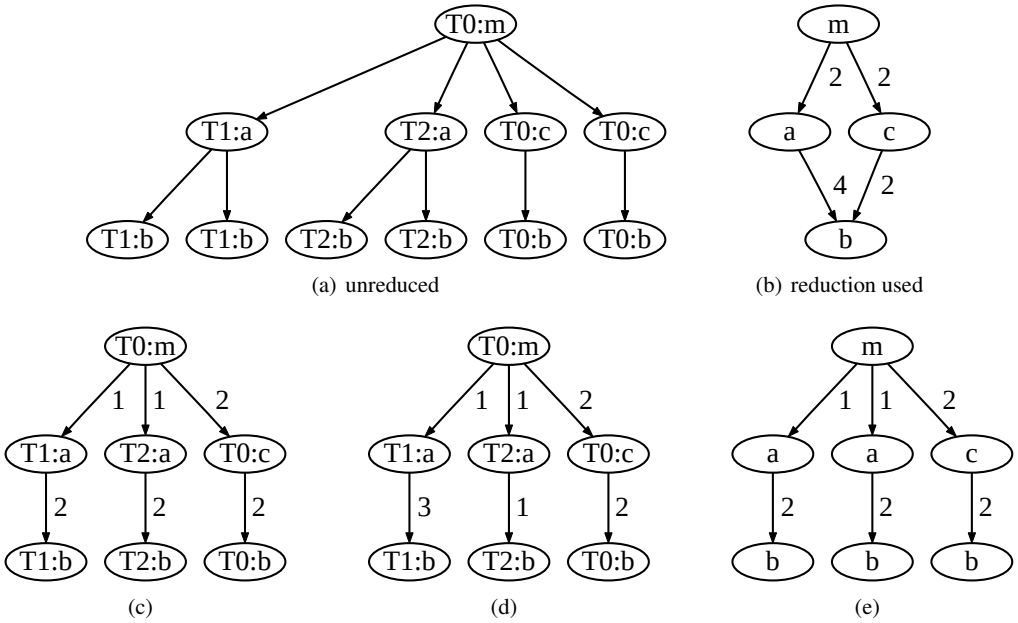

Figure 1. Example graphs illustrating alternative choices for call-graph representations.

method can be executed several times in more than one thread. Therefore, in unreduced call graphs, nodes are initially labeled with a prefix consisting of the respective thread ID and method name. Figure 1(a) illustrates an example of such a call graph, which shows all method calls of one program execution.

Reduced call-graph representations, by contrast, make the graph more compact. Figure 1(b) shows an example of such a representation, which is the reduced version of the call graph in Figure 1(a). Our approach employs this more concise *totally reduced* graph representation without thread IDs. Each method is uniquely represented by exactly one node that does not depend on a thread. We introduce edge weights as in [17] to summarize call frequencies: Every edge weight captures the total number of calls between the methods represented by the two connected nodes.

For the detailed call-graph representation, there are several tradeoffs to consider. In particular, we discuss the effects of temporal relationships, graph size, thread identification, and replicated tasks.

Including accurate temporal relationships in the graph representation may cause too much overhead. For the localization of defects in multithreaded software, it is natural to encode temporal information in call graphs, e.g., to tackle race conditions. The call graphs such as the one in Figure 1(a) do not encode any order of execution of the different threads and methods. One straightforward approach to include such information uses temporal edges, as done by Liu et al. [25]. The problem with this idea, however, is that the overhead to obtain such information can be large and would require sophisticated tracing techniques. Furthermore, such overhead may significantly influence program behavior – possibly making a failure disappear. In addition, increasing the amount of information in the call graph makes the graph mining process more difficult and time-consuming. We therefore employ a more lightweight approach without temporal information encoded in the graphs, as explained in Section 3.3.

Call graphs directly derived from program executions – such as the one in Figure 1(a) – become very large in practice. Even for a small program, the number of method calls can become so large that data-mining algorithms will not scale. Therefore, a compact representation is required. Figure 1(c) represents the *total reduction* of Figure 1(a), merging all nodes with the same node label. This reduction encodes some of the information that was previously contained in the graph structure in the edge weights.

DOI: 10.1002/cpe

Thread IDs differ between program executions. Figure 1(c) illustrates a call-graph representation that contains the thread IDs in the node labels. This is awkward, as threads are allocated dynamically by the runtime environment or the operating system. Therefore, various correct executions could lead to threads with different IDs for the same method call, even for a program using the same parameters and input data. We therefore would not be able to compare several program executions based on the node labels. Omitting this information from the graph in Figure 1(c) would directly result in the graph shown in Figure 1(e).

The effects of replicated tasks and varying thread interleavings must be addressed as well. Graphs such as the ones in Figures 1(c), (d) and (e) suffer from two problems: (1) They might contain a high degree of redundancy that does not help finding defects. For example, a program using thread pools could have a large number of threads with similar calls due to the execution of replicated tasks (and therefore similar method calls). This typically produces a call graph with several identical and large subtrees, where the information meaningful for defect localization is not evident. (2) The call frequencies (i.e., the edge weights) might not be useful for defect localization either. Different execution schedules of the same program can lead to graphs with widely differing edge weights. This effect can disturb data-mining analyses, as such differences do not have to bear any relationship to infections. To illustrate, consider method a in Figure 1(c) as the `run()` method that calls the worker task method b, which in turn takes work from a task pool. Occasionally, thread 1 and thread 2 would both call method b twice, as in Figure 1(c). In other cases as in Figure 1(d), depending on the schedules, thread 1 could call method b three times, while thread 2 would only call it once or vice versa.

3.3. Our Graph Representation

To avoid the pitfalls illustrated in the discussion so far, our approach employs a graph representation that avoids repeated substructures. We merge all nodes that refer to the same method into one single node. This approach leads to the graph representation presented in Figure 1(b). Our representation is robust in the sense that different schedules do not influence the graph structure. The reason is that methods executed in different threads are mapped to the same nodes. The downside of this representation is that graph structures from different executions rarely differ. Consequently, a structural analysis of the call graphs as in other approaches (e.g., [25, 26]) is less promising. To compensate this effect, we encode additional information in the edge weights. This addition has turned out to be helpful for discovering defective program behavior [17].

To generate call graphs for multithreaded applications, we employ AspectJ [29] and use it to weave tracing functionality into a program. AspectJ has been shown to be well-suited for program-trace generation and infection detection in multithreaded programs [30]. AspectJ introduces additional overhead and execution slowdowns; we observed a typical increase in execution time between 50% and 100% for the programs used in our evaluation (see Section 6).

4. LOCATING DEFECTS IN MULTICORE APPLICATIONS

In this section, we present a defect-localization procedure that presents developers with a ranking of defective methods, ordered by the probability of defects. Software developers can use this ranking to inspect method code, starting with the highest-ranked method. We present an overview of the defect-localization approach, followed by more details on our data-mining-based technique.

4.1. Overview

Algorithm 1 describes our general methodology for defect localization. The algorithm starts with a set T of traces obtained from program executions. A trace is an unreduced call graph where every method invocation leads to a new edge and a new node (see Figure 1(a)).

We employ a *test oracle* to decide whether a program execution is correct or not (Line 3 in Algorithm 1). Such oracles are specific for the examined program, and their purpose is to decide if a certain execution yields any observable problems (i.e., a *failure*). An observable problem can be a

DOI: 10.1002/cpe

wrong output or other erroneous behavior such as a deadlock. In this article, we assume that some kind of test oracle is available. In practice, many available testing benchmarks include test oracles.

Algorithm 1 Overview of call-graph-based defect localization.

Input: a set of program traces $t_j \in T$
Output: a method ranking based on each method's defect probability $P(m_i)$
1: $G = \emptyset$ // initialize a set of reduced graphs
2: **for all** traces $t_j \in T$ **do**
3: check if t_j was a correct execution and assign a $class \in \{correct, failing\}$ to t_j
4: $G = G \cup \{reduce(t_j)\}$
5: **end for**
6: calculate $P(m_i)$ for all Methods m_i in G

Using the oracle, our algorithm assigns a class ($correct$ or $failing$) to every trace $t_j \in T$. The algorithm then reduces every t_j to obtain a new call graph, which is assigned to the class of either correct or failing executions. Based on these graphs, the last step calculates for every method m_i its defect probability. This probability is used to rank potentially defective methods. The ranking is finally presented to the software developer.

4.2. Calculating Defect Probabilities

We now describe how to calculate the defect probability of a method (Line 6 in Algorithm 1) using data-mining techniques. The goal is to find out which methods in the call graph of a program discriminate best between correct and failing executions. We analyze the edge weights of the call graphs to derive such probabilities. Then we create a feature table containing all edges as columns and all program executions (represented by their reduced call graphs) as rows (see Table I).

For illustration, consider the example in Table I. The first column corresponds to the edge from method a to method b, the second column to the edge from a to c, the third column to the edge from b to d and the fourth column represents an edge from b to e. The last column contains the class ($correct$ or $failing$). The rows correspond to reduced call graphs $g_1, ..., g_n \in G$, which are derived from program executions. If a certain edge is not contained in a call graph, the respective cell is 0, i.e., there is no such method call. For example, no graph of $failing$ executions has edge $b \to e$.

	$a \to b$	$a \to c$	$b \to d$	$b \to e$	C
g_1	1	1	3	254	$correct$
g_2	1	1	3	12	$correct$
g_3	1	2	30	0	$failing$
g_4	1	2	30	0	$failing$
g_5	1	9	3	721	$correct$
g_6	1	1	3	54	$correct$
g_7	1	2	30	0	$failing$
$GainRatio$	0.00	0.68	1.00	1.00	
$InfoGain$	0.00	0.99	0.99	0.99	

Table I. Example of a feature table.

We analyze the edge weights of tables such as Table I with a feature-selection algorithm that calculates the strength of discrimination of each column, i.e., of each graph edge. The particular output is the *information-gain-ratio* measure ($GainRatio$, see Definition 1) for each column, which we obtain with the Weka machine-learning suite [31]. These numbers indicate which of the edges have the greatest decisive power to clearly classify a program run as correct or incorrect.

Definition 1 (Information-Gain Ratio, $GainRatio$)**.** *Let D be a table as in Table I. Each row $g_1, ..., g_n$ represents a program execution. C is one distinguished column in D that maps each*

DOI: 10.1002/cpe

execution to the class in $\{correct, failing\}$. $\mathbb{D}_C$ *is the domain of* C*, and* $D_{C=i}$ *denotes the set of rows that belong to the* i*-th class (*$i \in \mathbb{D}_C$*). Let* A *denote any other column different from* C*. We assume that the domain of* A *consists of numerical values. The* information-gain ratio *(*$GainRatio$*) [32] is a measure based on* information gain *(*$InfoGain$*) and* entropy *(*$Info$ *[32]). The* $GainRatio$ *measures the discriminativeness of an attribute* A *when values* $v \in A$ *partition the dataset* D*. The partitioning is done in a way that the* $GainRatio$ *of* A *is maximized. This requires a discretization of* A*'s values into* n *intervals (see, e.g., [33] for more information on the discretization), where* $\mathbb{D}_A$ *is the domain of the discrete intervals of* A *(*$n = |\mathbb{D}_A|$*).* $D_{A \in j}$ *is the set of rows of* D *that belong to the* j*-th interval of* A *(*$j \in \mathbb{D}_A$*).*

$$
\begin{aligned}
Info(D) &:= -\sum_{i \in \mathbb{D}_C} \frac{|D_{C=i}|}{|D|} \cdot \log_2\left(\frac{|D_{C=i}|}{|D|}\right) \\
InfoGain(A, D) &:= Info(D) - \sum_{j \in \mathbb{D}_A} \frac{|D_{A \in j}|}{|D|} \cdot Info(D_{A \in j}) \\
SplitInfo(A, D) &:= -\sum_{j \in \mathbb{D}_A} \frac{|D_{A \in j}|}{|D|} \cdot \log_2\left(\frac{|D_{A \in j}|}{|D|}\right) \\
GainRatio(A, D) &:= \frac{InfoGain(A, D)}{SplitInfo(A, D)}
\end{aligned}
$$

The $GainRatio$ measure is frequently used in data analysis, in particular in decision-tree induction [32]. The upper bound of the $GainRatio$ is 1. In this case an attribute discriminates perfectly between classes. At 0, the lower bound of $GainRatio$, an attribute has no influence on class discrimination.

Let us reconsider the execution of a program as in Table I to clarify how the $GainRatio$ measure locates defective methods.

Example 1. *In Table I, suppose that method* b *contains a defect that (1) affects* b*'s invocations of other methods and (2) affects the value of a global variable that is read in another method* a*. Suppose that methods are invoked as follows:*

- $a \to b$*: In each execution of the program, method* a *invokes method* b *once.*
- $a \to c$*: method* a *always invokes method* c *at least once. In addition, method* c *is called in a loop which has a condition affected by the values derived in method* b*. In failing executions, the loop terminates after exactly one iteration. In correct executions in turn, the loop is not executed, or it terminates after two or more iterations. So exactly two calls of method* c *can be used as a predictor for failing executions.*
- $b \to d$*: When method* b *calls method* d*, the number of calls is ten times higher in failing executions, due to the defect in method* b*.*
- $b \to e$*: The defect in method* b *causes method* e *not to be invoked at all in failing executions. In correct executions in turn, the number of invocations varies.*

The $GainRatio$ *value of each column is shown in the bottom rows of Table I and can be interpreted as follows: Columns* $b \to d$ *and* $b \to e$ *have the highest value and thus point to a defect in* b*, because* b *is the caller (i.e., on the left side of the arrow). Column* $a \to c$*'s value has a significantly increased* $GainRatio$*, as* a *reads infected values caused by method* b*. Column* $a \to b$ *is the only one with* $GainRatio = 0$ *and presumably no defect.*

The $InfoGain$ *values are less appropriate for defect localization, even though this measure is widely used for data mining. This is due to two reasons:*

1. *As the table shows, columns* $a \to c$*,* $b \to d$ *and* $b \to e$ *all have the same value, which makes a distinction more difficult. The column* $a \to c$ *whose methods do not contain a defect is not ranked lower than the other two columns. The explanation is that* $GainRatio$

DOI: 10.1002/cpe

normalizes the $InfoGain$ value by $SplitInfo$, which is the entropy of the discretization of the attribute into intervals. In our case, $a \to c$ has three intervals leading to the maximum $GainRatio$: $[1, 1.5), [1.5, 5.5), [5.5, 9]$, referring to $correct, failing, correct$, respectively. (The interval borders are chosen as the midpoint between the highest value from the lower interval and the lowest value from the higher interval, e.g., 1.5 is between 1 and 2.) This discretization allows to correctly identify the failing case (i.e., value 2; $2 \in [1.5, 5.5)$). The higher number of intervals of $a \to c$ (three instead of two for all other columns) leads to a higher $SplitInfo$. Consequently, $GainRatio$ is lower and reports a lower defect probability for column $a \to c$ than $InfoGain$.

2. *The maximum value of $InfoGain$ can only be 1 if the distribution of classes in C is equal. In this example in turn, the ratio of $correct$ to $failing$ executions is $4 : 3$. In skewed distributions of C, the $InfoGain$ can be misleadingly low, say, 0.47 (this refers to a ratio of $1 : 9$), even if an attribute can perfectly tell classes apart. $InfoGain$ is therefore less suited for software development scenarios, as values might be misleading, and defective programs typically do not have an equal probability of correct and failing executions.*

Besides $GainRatio$, we could have chosen from a number of other feature-selection algorithms. Our previous work (see Section 3.1), however, shows that algorithms based on *entropy* are more appropriate and locate defects well.

So far, we have derived defect probabilities for every column in the table, i.e., for edges. However, we are interested in defect probabilities for methods. As a method can call several other methods, we assign every column to the calling method. More specifically, we calculate the method probability $P(m_i)$ as the maximum of the gain-ratio values of the columns assigned to method m_i. We use the maximum because it refers to the most suspicious invocation of a method. Other invocations are less important, as they might not be related to a defect. However, the information which specific invocation within method m_i is most suspicious (the column with the highest probability) can be important for a software developer to find and fix the defect. This information is additionally reported by our tool.

5. A DETAILED EXAMPLE

We illustrate a typical defect and the process of identifying its location with our approach. The following example utilizes excerpts from the *GarageManager* program [34] that we use in our later evaluation.

The defect. The calculation of the `taskNumber` variable can produce a negative value, which is read in method `GoToWork()` (see Listing 1) to calculate its modulo-8 value. This value is then passed into a `switch-case` block. This block, however, expects values between 0 and 7. Negative values can result when Java calculates the modulo operation on a negative number. There are two positions where a developer can choose to modify the code to fix the bug: (1) The `switch-case` block, by adding negative cases or a default case; (2) The parts of the source code where `taskNumber` is calculated (method `SetTaskToWorker()`).

From the defect to an infection. Figure 2 illustrates the reduced call-graph representation for a case when the *GarageManager* program fails. As mentioned in Section 3.2, nodes represent program methods and the edge weights capture the total call frequency. For example in the execution depicted in Figure 2, the `main` method calls the `GiveTasksToWorkers` method once; note that our representation abstracts from the particular thread ID.

In detail, this call graph reveals that executing `run()` generates additional threads. In particular, there are four "worker" threads each calling methods `WaitForManager()`, `GoToWork()` and `PrintCard()` and one "manager" thread calling the remaining methods. In `WorkingOn()` (a defective method), the program state becomes infected: Three threads evaluate their switch statement to 0, 1 and 7, but the fourth thread has a negative value, thus causing the thread not to call any further methods (these details of the example execution cannot be seen in the graph, only the effects on the control structure are visible, i.e., the call of methods).

DOI: 10.1002/cpe

```
1 switch (taskNumber % 8) {
2   case 0: WorkingOn("Cleaning", 1000);
3           break;
4           // omitting similar cases 1 to 5...
5   case 6: WorkingOn("Working on breaks", 2200);
6           break;
7   case 7: WorkingOn("Fixing engines", 2400);
8           break;
9 }
```

Listing 1: Method `void GoToWork()` of the *GarageManager* program (excerpt).

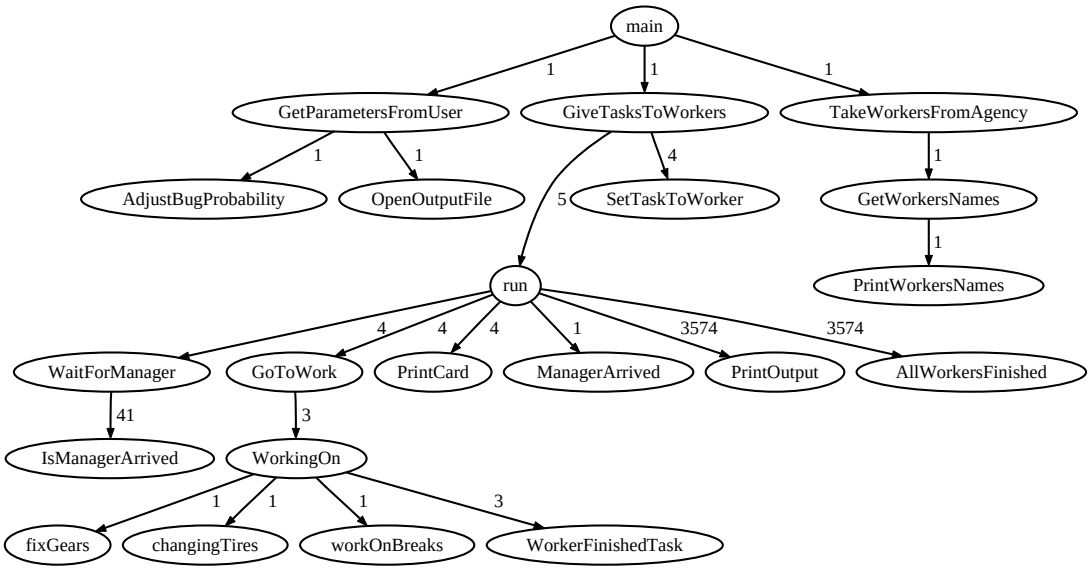

Figure 2. Call graph from a failing *GarageManager* execution.

From an infection to a failure. The aforementioned infection causes the fourth thread not to call `WorkerFinishedTask()`. This method decreases a variable of the global `status` object. This object is queried by `AllWorkersFinished()` in method `run()` (see Listing 2). `AllWorkersFinished()` will never be `true`, as `status` will always indicate that only three out of four "worker" threads have finished their tasks. This causes an infinite loop in `run()`. We manually stopped the loop after 3,574 iterations. In other words, the infection has caused a deadlock, an observable program behavior that is a failure.

Locating the defect. In our experiments, our approach has found the three methods `GoToWork()`, `WorkingOn()` and `run()` (ordered by increasing ranking position) to have the highest defect probabilities. Thus, the defect was pinpointed directly. The high defect probability for `WorkingOn()` is due to a follow-up infection, as it is always called from `GoToWork()`. The `run()` method has a high defect probability as well, caused by the huge number of method calls in the infinite loop, compared to correct executions. Both methods are inherently connected to the same defect.

6. EXPERIMENTAL EVALUATION AND COMPARISION

We present detailed experimental results to validate our approach and show that it works in practice on real programs. This section describes the benchmark programs and their defects, the experimental

DOI: 10.1002/cpe

```
1 synchronized (status) {
2   System.out.println("Manager arrived !");
3   status.ManagerArrived();
4 }
5 boolean tasksNotFinished = true, printedOutput = false;
6 while (tasksNotFinished) {
7   printedOutput = PrintOutput(printedOutput);
8   synchronized (status) {
9     if (status.AllWorkersFinished())
10      tasksNotFinished = false;
11    else
12      yield();
13  }
14 }
```

Listing 2: Method `void run()` (excerpt).

setting and the metrics used to interpret the results. Finally, we compare our method to related techniques.

6.1. Benchmark Programs and Defects

Our benchmark contains a range of different multithreaded programs. The benchmark covers a broad range of tasks, from basic sorting algorithms and various client-server settings to memory allocators, which are fundamental constructs in many programs [35]. All benchmark programs are written in Java. Our bug detection tool is developed for Java programs and is based on AspectJ, which requires Java programs to work.

Most of the benchmark programs have been used in previous studies [34]. We have slightly modified some of the applications; for example, in the *GarageManager* application, we replaced various text output statements with methods that contain code simulating the assignment of work to different tasks. Furthermore, we have included in our benchmark two typical client-server applications from the open-source community, which represent an important class of real applications.

Table II lists all programs, their size in terms of methods, and their normalized lines of code (LOC)[†].

Program	#M	LOC	#T	Source	Description
AllocationVector (*Test*)	6	133	2	[34]	Allocation of memory
GarageManager	30	282	4	[34]	Simulation of a garage
Liveness (*BugGen*)	8	120	100	[34]	Client-server simulation
MergeSort	11	201	4	[34]	Recursive sorting implementation
ThreadTest	12	101	50	[34]	CPU benchmark (random divisions)
Tornado	122	632	100	[36]	HTTP Server
Weblech	88	802	10	[37]	Website download/mirror tool

Table II. Multithreaded benchmark programs (#M/#T is the number of methods/threads).

Table III shows additional details on the complexity of the programs, including statistics on nested block depth of methods, depth of inheritance tree, and method cyclomatic complexity [38]. All programs have about the same inheritance depth. In programs with a high cyclomatic complexity,

[†]We always use the sum of non-blank and non-comment LOC inside method bodies.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

humans tend to have a much more difficult time finding bugs, which is why they are good for assessing debugging tools.

Program	Method Nested Block Depth avg.	Method Nested Block Depth max.	Max. Depth of Inheritance Tree	Method Cyclomatic Complexity avg.	Method Cyclomatic Complexity max.
AllocationVector (*Test*)	2.75	5	2	5.00	14
GarageManager	1.71	5	2	2.97	11
Liveness (*BugGen*)	1.91	5	2	2.73	8
MergeSort	1.60	3	2	3.33	8
ThreadTest	1.79	4	3	2.21	6
Tornado	1.42	5	3	1.73	12
Weblech	1.59	7	2	2.22	15

Table III. Additional complexity statistics of the multithreaded benchmark programs.

For evaluation purposes, all benchmark programs have intentionally seeded defects that are known and documented. All defects are representatives of common multithreaded programming errors. The defects cover a broad range of error patterns, such as atomicity violations/race conditions, on one or several correlated variables, deadlocks, but also other kinds of programming errors, e.g., originating from non-parallel constructs that may influence parallel program behavior.

6.2. Defect Patterns

Based on the classification of Farchi et al. [2], we categorize our defect pattern in the following list and show code outlines to illustrate some of the most important defects.

(1) *AllocationVector*; defect pattern: **"two-stage access"**. Two steps of finding and allocating blocks for memory access are not executed atomically, even though the individual steps are synchronized. Thus, two threads might allocate the same memory and cause incorrect interference. In the code excerpt, this can happen in Line 3 – another thread could get the same information from the `getFreeBlockIndex()` function at a certain point in time before `markAsAllocatedBlock()` is called.

```
1 for (int i = 0; i < resultBuf.length; i++) {
2   resultBuf[i] = vector.getFreeBlockIndex();
3   // everything can happen here...
4   if (resultBuf[i] != -1) {
5     vector.markAsAllocatedBlock(resultBuf[i]);
6   }
7 }
```

Listing 3: Excerpt of the *AllocationVector* program illustrating a "two-stage access" defect.

(2) *GarageManager*; defect pattern: **"blocking critical section"**. The defect itself is a combination of an incorrectly calculated value due to a forgotten switch case. When this situation occurs, no task is assigned to a particular thread, while a global variable is treated as if work had been assigned. Thus, fewer threads than the number of threads recorded as active are active. This causes the program to deadlock. The *GarageManager* defect pattern has been discussed in Section 5.

(3) *Liveness*; defect pattern: similar to the **"orphaned thread"** pattern. When the maximum number of clients is reached, the next requesting client is added to a stack. Although this data structure and a global counter are synchronized, it can happen that the server becomes available while the client is added to the stack (i.e., the server becomes available immediately after the `if`

DOI: 10.1002/cpe

condition has been checked in Line 8). In this case, the client will never resume and will not finish its task.

```
1 synchronized (actualUsers) {
2   if (actualUsers.get() < maxAllowedUsers) {
3     actualUsers.inc();
4     accessGranted = true;
5   }
6 }
7 // ...
8 if (!accessGranted) {
9   // accessGranted might change!
10  synchronized (suspendedClients) {
11    suspendedClients.add(this);
12  }
13  this.suspend();
14 }
```

Listing 4: Excerpt of the *Liveness* program illustrating an "orphaned thread" defect.

(4) *MergeSort*; defect pattern: **"two-stage access"**. Although methods working on global thread counters are synchronized, the variables themselves are not, which might lead to atomicity violations. In particular, threads ask how many subthreads they are allowed to generate. When two threads apply at the same time, more threads than allowed are generated. This can lead to situations where parts of the data are not sorted. In particular, the `AvailableThreadsState()` could change immediately after the check of the `switch` condition in Line 1.

```
1 switch (AvailableThreadsState()) {
2   case 1:
3   leftSon.start();
4   DecreaseThreadCounter();
5   rightSon.Sorting();
6   // ...
7 }
```

Listing 5: Excerpt of the *MergeSort* program illustrating a "two-stage access" defect.

(5) *ThreadTest*; defect pattern: **"blocking critical section"**. The generation of new threads and checking a global variable for the maximum number of threads currently available is not done correctly in case of exceptions, which occur randomly in *ThreadTest*, due to divisions by zero. This leads to a deadlock when all threads encounter this situation. We classify an execution as failing when at least one thread encounters this problem, due to reduced performance. In the code excerpt, the thread terminates without having called `finalizeWork()`. This can happen when an exception is triggered in Line 3. This causes the control flow to bypass the execution of Line 4 and resume execution at Line 6. In this case, the thread will not work on further tasks.

(6) *Tornado*; defect pattern: **"no lock"**. Synchronization statements are removed in one method (Lines 2 and 6). This leads to a race condition and ultimately, in the context of *Tornado*, to unanswered HTTP requests.

DOI: 10.1002/cpe

```
1 try {
2   for (int i = 0; i < ITERATE; ++i)
3     result = i / (int) (Math.random() * factor);
4   finalizeWork();
5 } catch (java.lang.ArithmeticException e) {
6   // ...
7 }
```

Listing 6: Excerpt of the *ThreadTest* program illustrating a "blocking critical section" defect.

```
1 public void dispatch(Socket socket) {
2 //   synchronized (taskPool) { // line removed
3       taskPool.add(socket);
4       taskPool.notify();
5       incrementBusyThreads();
6 //   }                          // line removed
7 }
```

Listing 7: Excerpt of the *Tornado* program illustrating a "no lock" defect.

(7) *Weblech*; defect pattern: **"no lock"**. Removed synchronization statements as in *Tornado*, resulting in Web pages that are not downloaded.

Regarding the *Weblech* program, we have two versions: *Weblech.orig* and *Weblech.inj*. In *Weblech.inj*, we have introduced a defect in method `run()` by removing all `synchronized` statements (Listing 8 shows an excerpt of this method with one such statement), aiming to simulate a typical programming error. During our experiments, we realized that the original non-injected version (*Weblech.orig*) led to failures in very rare cases as well. The failure occurred in only 5 out of 5,000 executions; we used a sample of the correct executions in the experiments. Thus, *Weblech.inj* contains the original defect besides the injected ones. With our tool, we were able to locate the real defect by investigating two methods only. The result is that two global unsynchronized variables (`downloadsInProgress` and `running`) are modified in `run()` (Lines 6 and 10 in the code excerpt), occasionally causing race conditions. To fix the defect in order to produce a defect-free reference, we added the `volatile` keyword to the variable declaration in the class header.

```
1 while ((queueSize() > 0 || downloadsInProgress > 0)
2   && quit == false) {
3   // ...
4   synchronized (queue) {
5     nextURL = queue.getNextInQueue();
6     downloadsInProgress++;
7   }
8   // ...
9 }
10 running--;
```

Listing 8: Excerpt of the `void weblech.spider.run()` method in the *Weblech* program illustrating a "no lock" defect.

DOI: 10.1002/cpe

6.3. Experimental Setting

Number of executions. Our defect-localization technique requires that we execute every program several times and that we ensure that there are sufficiently many examples for correct and failing executions. This is necessary since we focus on occasional bugs (see Section 3.1), i.e., failures whose occurrence depends on input data, random components or non-deterministic thread interleavings. Furthermore, we aim to achieve stable results, i.e., analyzing more executions would not lead to significant changes. We used this criterion to determine the number of executions required, in addition to obtaining enough correct and failing cases. Table IV in Section 6.5 will summarize the number of correct and failing executions for each benchmark program.

Varying execution traces. In order to obtain different execution traces from the same program, we rely on the original test cases that are provided in the benchmark suite. *MergeSort*, for instance, comes with a generator creating random arrays as input data. Some programs have an internal random component as part of the program logic, i.e., they automatically lead to varying executions. *GarageManager*, for instance, simulates varying processes in a garage. Other programs produce different executions due to different thread interleavings that can lead to observable failures occasionally. For the two open-source programs, we constructed typical test cases ourselves; for the *Tornado* Web server, we start a number of scripts simultaneously downloading files from the server. For *Weblech*, we download a number of files from a (defect-free) Web server.

Test oracles. We use individual test oracles that come with every benchmark program. For the two open-source programs, we compose test oracles that automatically compare the actual output of a program to the expected one. For example, we compare the files downloaded with *Weblech* to the original ones.

Testing environment. We run all experiments on a standard HP workstation with an AMD Athlon 64 X2 dual-core processor 4800+. We employed a standard Sun Java 6 virtual machine on Microsoft Windows XP. In the evaluation, our graph graph representation exhibits another advantage: As the graph structure is independent of the particular scheduling and number of threads (which are modeled as edge weights), the call graphs would have the same structure on different machines, provided that tests use the same inputs and the control flow follows the same paths.

6.4. Accuracy Measures for Defect-Localization Results

First of all, the locations of the actual defects are known, so the report of a method containing a defect can be directly compared to the known location to see if this is true or not. If there is more than one location which can be altered to fix a defect, we refer to the position of the first of such methods in the ranking. For cases as in *Weblech.orig* where the defect can be fixed outside a method body (e.g., in the class header), one can still identify methods that can be altered to fix the erroneous behavior.

Our experiments produce ordered lists of methods. To evaluate the accuracy of the results, we report the position of the defective method in such a list. This ranking position corresponds to the number of methods a software developer has to inspect in order to find the defect. If two or more methods have the same defect probability, we use a second static ranking criterion: We sort the methods with the same defect probability by decreasing LOC size. Previous research has shown that the LOC size frequently positively correlates with the defect probability of a method being defective [39]. In order to estimate the effort to find a defect, we compare the ranking position with the total number of methods in a program. In addition to the ranking, our tool also provides more fine-grained information, such as the suspected call within a method.

Another quality criterion is the comparison of our method with the expected value for manual defect localization; in the manual approach, one would expect to find the defect after reviewing about half of the program methods.

DOI: 10.1002/cpe

As method sizes can vary significantly, it is sometimes more appropriate to consider the LOC rather than only the number of methods involved. Our tool therefore shows the percentage of LOC to review as an addition to the ranking position. This percentage is calculated as the ratio of methods that has to be considered in the program, i.e., the sum of LOC of all methods that have a ranking position lower than or equal to the position reported in the table, divided by the total LOC (see Table II).

The scoring method used to measure the accuracy of defect localization in this paper follows related work in this area. It has originally been suggested by Renieris and Reiss [40]. Since then, this method has been adopted in many studies, e.g., in [13, 14, 26, 41]. It has its advantages compared to methods relying on the precision/recall scheme (e.g., [25, 28]), which stem from the fact that it does not only measure whether a defect is successfully located or not, but also how expensive it is to locate a defect.

6.5. Experimental Results

We present evaluation results in Table IV, which illustrate for each benchmark program how effective our method has been to find a defective method and how much code reading could be saved during the code inspections to locate the defect. Columns (1) and (2) show the number of program executions as explained in Section 6.3. Column (3) shows the ranking position for the method that contains a previously implanted defect, as produced by our debugging technique. Column (4) shows the maximum lines of code that a developer would have to review in the worst case to locate the defect. Column (5) in turn shows the actual lines of code that need to be reviewed to locate the bug. Column (6) shows the corresponding reduction of lines of code to review, i.e., the fraction of lines that a developer does not have to inspect if he or she uses our technique.

In all five out of eight programs, the defective method is ranked first. It has a low rank only in one program (Tornado). This is because this program is complex and has more methods than the others. However, the quantitative comparison shows that the reduction of code to review is in the same range as with the other programs.

	Executions		Defect-localization Metrics				Improvement
Program	(1) # correct	(2) # failing	(3) Defect Ranking Position	(4) Maximum LOC to Review	(5) Actual LOC to Review	(6) Actual %LOC to Review	(7) Reduction of LOC to Review
AllocationVector	383	117	1	133	23	17.3%	82.7%
GarageManager	74	26	1	282	40	14.2%	85.8%
Liveness	149	53	1	120	53	44.2%	55.8%
MergeSort	668	332	1	201	52	25.9%	74.1%
ThreadTest	207	193	1	101	19	18.8%	81.2%
Tornado	362	8	14	632	147	23.3%	76.7%
Weblech.orig	494	5	2	802	187	23.3%	76.7%
Weblech.inj	985	15	5	802	175	21.8%	78.2%

Table IV. Defect-localization results.

Considering averages, the methods containing the seeded defects rank at position 3.3. Looking at Table II, one program consists of 45.6 methods on average (counting *Weblech* twice). Taking this information into account, investigating 3.3 methods on average corresponds to 7.1% of all methods (or equivalently, 23.6% of the code) a developer has to review maximally to find the defects. This is low. In other words, a developer has to consider less than a quarter of the source code of our programs in order to find a defect in the worst case. This reduces the percentage of methods (code)

DOI: 10.1002/cpe

to review by a factor of seven (code: more than by half) when compared to an average expected amount of 50% of methods (code) to review. Note that these all values are obtained without any prior knowledge of the code, which might further narrow down the locations to be inspected. Furthermore, these are worst-case maximum values, for two reasons: (1) Usually not all lines of a method need to be inspected, in particular due to information reported additionally by our tool which call within a method is the most suspicious one. (2) The methods ranked frequently at the top are good hints for a defect, even if the defective method itself is in some lower ranks. This heuristic is based on our experience that non-defective methods ranked at the top are often in the vicinity of the defective method, i.e., they might be invoked from the defective method.

Figure 3 provides an illustration of the percentage of located defects versus the percentage of source code that does not need to be examined for all programs in the evaluation. This graphing technique is commonly used (e.g., in [13, 14, 26, 41]) to visualize the effectiveness of a defect-localization technique. In our case, it shows that we can skip the inspection of 50% of the code and still find 100% of the defects. If we skip inspecting 70%, we would still find more than 80% of the defects. This is a significant gain in programmer productivity.

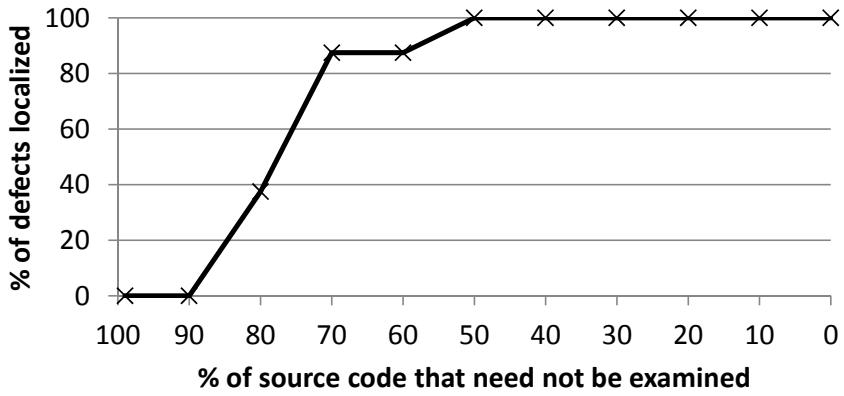

Figure 3. The percentage of defects located in the entire benchmark while skipping inspection of a certain percentage of source code.

6.6. Result Comparisons with Related Work

We now compare our approach with two techniques from related work that are comparable in that they can produce lists of defective methods.

Our experiments with the IBM MulticoreSDK [8] applied to all programs used in our evaluation (see Section 6) reveal that it is not able to find any of the defects. From the eight benchmark programs, the MulticoreSDK incorrectly classified seven of them as defect-free. For the last one, it incorrectly generated a false-positive warning.

We applied FindBugs [21] to all programs in our benchmark. FindBugs did not directly report any of the defects. At the same time, FindBugs produces false-positive warnings: On average, there are 5.8 warnings per program that on average affect 4.5 different methods. The warnings refer to the correct method names in just four out of eight programs. Further, the warnings are not prioritized, so a developer would have to inspect the entire code of all methods with warnings. In each of the four programs, inspection amounts to 47.5%, 36.8%, 29.2% and 29.2% of the source code, respectively. If FindBugs was improved by a method ranking technique, such as inspecting larger methods first (as in this article), then developers could save time finding the respective defects and reduce the amount of reviewed code to 14.2%, 25.9%, 25.4% and 25.4%, respectively. In contrast, inspecting up to 25.9% of the source code with our technique finds seven out of the eight defects (see the last column in Table IV). These results are better than FindBugs. Compared to our approach, FindBugs does not offer the developer any hint on finding the remaining four defects, as they are not reported at all.

DOI: 10.1002/cpe

7. LONG-TERM DIRECTIONS FOR DATA MINING FOR PARALLEL APPLICATION DEFECTS

Motivated by the promising results presented in this article, we outline long-term directions for the emerging field of data-mining-based defect localization. In general, we believe that parallel program debugging with data-mining techniques has the potential to become a larger research area in its own right.

The technique in this article exploits control-flow information in a program execution. Adding information about the data flow can increase the defect-localization accuracy. Cheng et al. [28] have identified that current call-graph-based defect-localization techniques are agnostic to defects that influence the data flow. This observation also applies to our study. Our own work on sequential programs [42] demonstrates that introducing data-flow information into call graphs is beneficial; defects can be located that have a greater influence on data flow than on control flow. We are currently working on an extension of our call-graph representation for multithreaded programs to include information from the data flow. This approach will make race detection more accurate. This is because unsynchronized threads incorrectly alter data and affect the values in the data flow in typical race situations.

Debugging techniques such as the one presented in this article execute a multithreaded program several times. In general, the specific thread interleavings that occur during execution are non-deterministic and are influenced by the operating-system scheduler, so we might require large numbers of executions to observe failures in rare interleavings. Additional control over thread schedules can improve our effectiveness in several ways. Firstly, executing thread schedules that are more likely to make defects manifest themselves (e.g., a race condition) can help reduce the number of repeated executions of a program. Secondly, thread schedules of correct and failing executions can be mutated using operators as in evolutionary algorithms [43] to search for similar behavior that leads to correct or failing executions. This approach directs the search to more promising locations in a large search space. Thirdly, program configurations for failing executions and their associated thread schedules can be employed for regression testing. In practice, regression tests are especially helpful when programmers add new functionality to a program but need to ensure that the existing features still work. Quality assurance can become more efficient if developers can reuse test cases consisting of failing program configurations and reproduce the exact thread interleaving that has caused an error to manifest itself. Based on insights from [6, 44] we are currently extending our data-mining technique in this direction.

Section 3.2 has discussed various representations of call graphs for multithreaded programs. Due to a number of issues related to multithreaded executions, we have opted for and deployed a relatively simple *total-reduction* graph representation. However, we believe that alternatives with more sophisticated graph representations are worth being investigated. This is motivated by our experiments with single-threaded programs [17], where graphs more sophisticated than the totally reduced ones have given way to a defect localization that is more precise. Graph representations can, for instance, include additional information on thread IDs as well as information on the temporal order of methods executed, similarly to [25, 26] for the single-threaded case. A possible solution we see for the problem of indeterministic thread IDs (see Section 3.2) is the introduction of thread classes. Each of these classes stands for a source-code context, i.e., a position in the source code where new threads are created. As an example, one class could stand for GUI-related threads and one for database-access-related threads. Further information to enhance the expressiveness of call graphs could be information on locks on certain objects. This information could be included as an annotation of nodes or edges.

Call graphs with information additional to the one contained in the total-reduction graphs require new and more elaborate analysis techniques. In [17] we have presented a technique for single-threaded programs that relies on frequent subgraph mining [45]. There, we first aim at finding subgraph structures that occur more frequently in failing executions before we apply a technique similar to the one described in Section 4. Finally, we combine two kinds of evidence. We plan to

DOI: 10.1002/cpe

extend the technique in [17] for multithreaded programs to identify suspicious structures in call graphs with more detailed annotations.

8. CONCLUSION

Debugging multithreaded software is difficult, and developers depend on effective tools for quality assurance. This article shows that data mining on call graphs is an effective approach to detect a wide range of errors that affect parallel program behavior. Such errors include race conditions, deadlocks and errors originating from the wrong usage of non-parallel language constructs. An additional advantage of our proposal is that this wide range of errors can be detected with one single technique. Our evaluations show that developers can skip the inspection of at least 50% of the code and still find 100% of the defects. On average only 7.1% of all program methods have to be investigated to find a defect. Significant effort can thus be saved during defect localization in multithreaded programs.

ACKNOWLEDGMENTS

We thank Alexander Bieleš, who has helped us with the implementation and the experiments, and Philipp W. L. Große for discussions on the graph representations and program instrumentations. Shmuel Ur has provided us with the defect benchmark [34]. We also thank the Excellence Initiative at KIT and the Landesstiftung Baden-Württemberg for their support.

REFERENCES

1. Pankratius V. Software Engineering in the Era of Parallelism. *Emerging Research Directions in Computer Science – Contributions from the Young Informatics Faculty in Karlsruhe*, Pankratius V, Kounev S (eds.). KIT Scientific Publishing, 2010; 45–52.
2. Farchi E, Nir Y, Ur S. Concurrent Bug Patterns and How to Test Them. *Proceedings of the 1st Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, 2003, doi:10.1109/IPDPS.2003.1213511.
3. Lu S, Park S, Seo E, Zhou Y. Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics. *SIGARCH Computer Architecture News* 2008; **36**(1):329–339, doi:2410.1145/1353534.1346323.
4. Engler D, Ashcraft K. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003, doi:10.1145/945445.945468.
5. Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R. Extended Static Checking for Java. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002, doi:10.1145/512529.512558.
6. Musuvathi M, Qadeer S, Ball T. CHESS: A Systematic Testing Tool for Concurrent Software. *Technical Report MSR-TR-2007-149*, Microsoft Research 2007.
7. O'Callahan R, Choi JD. Hybrid Dynamic Data Race Detection. *SIGPLAN Notices* 2003; **38**(10):167–178, doi: 10.1145/966049.781528.
8. Qi Y, Das R, Luo ZD, Trotter M. MulticoreSDK: A Practical and Efficient Data Race Detector for Real-World Applications. *Proceedings of the 7th Workshop on Parallel and Distributed Systems (PADTAD)*, 2009, doi: 2410.1145/1639622.1639627.
9. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems* 1997; **15**(4):391–411, doi:10.1145/265924.265927.
10. Tzoref R, Ur S, Yom-Tov E. Instrumenting Where it Hurts – An Automatic Concurrent Debugging Technique. *Proceedings of the 16th International Symposium on Software Testing and Analysis (ISSTA)*, 2007, doi:2410.1145/1273463.1273469.
11. Zeller A. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., 2009.
12. Eichinger F. Data-Mining Techniques for Call-Graph-Based Software-Defect Localisation. PhD Thesis, Department of Informatics, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany May 2011.
13. Liu C, Fei L, Yan X, Han J, Midkiff SP. Statistical Debugging: A Hypothesis Testing-Based Approach. *IEEE Transactions on Software Engineering* 2006; **32**(10):831–848, doi:10.1109/TSE.2006.105.
14. Jones JA, Harrold MJ. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005, doi:10.1145/1101908.1101949.
15. Jones JA, Harrold MJ, Stasko J. Visualization of Test Information to Assist Fault Localization. *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, 2002, doi:2410.1145/581339.581397.

16. Abreu R, Zoeteweij P, Golsteijn R, van Gemund AJ. A Practical Evaluation of Spectrum-Based Fault Localization. *Journal of Systems and Software* 2009; **82**(11):1780–1792, doi:10.1016/j.jss.2009.06.035.
17. Eichinger F, Böhm K, Huber M. Mining Edge-Weighted Call Graphs to Localise Software Bugs. *Proceedings of the 8th European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2008, doi:10.1007/978-3-540-87479-9_40.
18. Eichinger F, Pankratius V, Große PWL, Böhm K. Localizing Defects in Multithreaded Programs by Mining Dynamic Call Graphs. *Proceedings of the 5th Testing: Academic and Industrial Conference – Practice and Research Techniques (TAIC PART)*, 2010, doi:10.1007/978-3-642-15585-7_7.
19. Gait J. A Probe Effect in Concurrent Programs. *Software: Practice and Experience* 1986; **16**(3):225–233, doi: 10.1002/spe.4380160304.
20. Zeller A. Yesterday, my Program Worked. Today, it Does Not. Why? *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 1999, doi:10.1007/3-540-48166-4_16.
21. Ayewah N, Hovemeyer D, Morgenthaler JD, Penix J, Pugh W. Using Static Analysis to Find Bugs. *IEEE Software* 2008; **25**(5):22–29, doi:10.1109/MS.2008.130.
22. Ayewah N, Pugh W. The Google FindBugs Fixit. *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, 2010, doi:10.1145/1831708.1831738.
23. Rutar N, Almazan CB, Foster JS. A Comparison of Bug Finding Tools for Java. *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE)*, 2004, doi:10.1109/ISSRE.2004.1.
24. Dallmeier V, Lindig C, Zeller A. Lightweight Defect Localization for Java. *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, 2005, doi:10.1007/11531142_23.
25. Liu C, Yan X, Yu H, Han J, Yu PS. Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. *Proceedings of the 5th SIAM International Conference on Data Mining (SDM)*, 2005.
26. Di Fatta G, Leue S, Stegantova E. Discriminative Pattern Mining in Software Fault Detection. *Proceedings of the 3rd International Workshop on Software Quality Assurance (SOQUA)*, 2006, doi:10.1145/1188895.1188910.
27. Eichinger F, Böhm K. Software-Bug Localization with Graph Mining. *Managing and Mining Graph Data*, *Advances in Database Systems*, vol. 40, Aggarwal CC, Wang H (eds.). chap. 17, Springer, 2010; 515–546, doi: 10.1007/978-1-4419-6045-0_17.
28. Cheng H, Lo D, Zhou Y, Wang X, Yan X. Identifying Bug Signatures Using Discriminative Graph Mining. *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, 2009, doi:10.1145/1572272.1572290.
29. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An Overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, 2001, doi:10.1007/3-540-45337-7_18.
30. Copty S, Ur S. Multi-threaded Testing with AOP Is Easy, and It Finds Bugs! *Proceedings of the 11th International Euro-Par Conference*, 2005, doi:10.1007/11549468_81.
31. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH. The WEKA Data Mining Software: An Update. *SIGKDD Explorations Newsletter* 2009; **11**(1):10–18, doi:10.1145/1656274.1656278.
32. Quinlan JR. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.
33. Elomaa T, Rousu J. Efficient Multisplitting on Numerical Data. *Proceedings of the 1st European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD)*, 1997, doi:10.1007/3-540-63223-9_117.
34. Eytani Y, Ur S. Compiling a Benchmark of Documented Multi-Threaded Bugs. *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004, doi:10.1109/IPDPS.2004.1303339.
35. Berger ED, McKinley KS, Blumofe RD, Wilson PR. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGPLAN Notices* 2000; **35**(11):117–128, doi:10.1145/356989.357000.
36. Tornado HTTP Server, software available at `http://tornado.sourceforge.net/`.
37. WebLech URL Spider, software available at `http://weblech.sourceforge.net/`.
38. McCabe TJ. A Complexity Measure. *IEEE Transactions on Software Engineering* 1976; **SE-2**(4):308–320, doi: 10.1109/TSE.1976.233837.
39. Nagappan N, Ball T, Zeller A. Mining Metrics to Predict Component Failures. *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006, doi:10.1145/1134285.1134349.
40. Renieres M, Reiss S. Fault Localization with Nearest Neighbor Queries. *Proceedings of the 18th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2003, doi:10.1109/ASE.2003.1240292.
41. Cleve H, Zeller A. Locating Causes of Program Failures. *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005, doi:10.1145/1062455.1062522.
42. Eichinger F, Krogmann K, Klug R, Böhm K. Software-Defect Localisation by Mining Dataflow-Enabled Call Graphs. *Proceedings of the 10th European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2010, doi:10.1007/978-3-642-15880-3_33.
43. Eiben AE, Smith JE. *Introduction to Evolutionary Computing*. Springer, 2003.
44. Choi JD, Zeller A. Isolating Failure Inducing Thread Schedules. *Proceedings of the 11th International Symposium on Software Testing and Analysis (ISSTA)*, 2002, doi:2410.1145/566172.566211.
45. Yan X, Han J. Discovery of Frequent Substructures. *Mining Graph Data*, Cook DJ, Holder LB (eds.). chap. 5, John Wiley & Sons, 2006; 99–115, doi:10.1002/9780470073049.ch5.