On Accelerating Source Code Analysis At Massive Scale

Ganesha Upadhyaya
Iowa State University, ganeshau@iastate.edu

Hridesh Rajan
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports
Part of the Computational Engineering Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/383

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
On Accelerating Source Code Analysis At Massive Scale

Abstract
Encouraged by the success of data-driven software engineering (SE) techniques that have found numerous applications e.g. in defect prediction, specification inference, etc, the demand for mining and analyzing source code repositories at scale has significantly increased. However, analyzing source code at scale remains expensive to the extent that data-driven solutions to certain SE problems are beyond our reach today. Extant techniques have focussed on leveraging distributed computing to solve this problem, but with a concomitant increase in computational resource needs. This work proposes a technique that reduces the amount of computation performed by the ultra-large-scale source code mining task. Our key idea is to analyze the mining task to identify and remove the irrelevant portions of the source code, prior to running the mining task. We show a realization of our insight for mining and analyzing massive collections of control flow graphs of source codes. Our evaluation using 16 classical control-/data-flow analyses that are typical components of mining tasks and 7 Million CFGs shows that our technique can achieve on average 40% reduction in the task computation time. Our case studies demonstrates the applicability of our technique to massive scale source code mining tasks.

Keywords
Source code analysis, Mining software repositories, Data-driven software engineering

Disciplines
Computational Engineering

This article is available at Iowa State University Digital Repository: http://lib.dr.iastate.edu/cs_techreports/383
On Accelerating Source Code Analysis At Massive Scale

Ganesha Upadhyaya

Hridesh Rajan

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Computational Engineering Commons
On Accelerating Source Code Analysis At Massive Scale

Ganesh Upadhyaya, Member, IEEE, and Hridesh Rajan, Member, IEEE

Abstract—Encouraged by the success of data-driven software engineering (SE) techniques that have found numerous applications e.g., in defect prediction, specification inference, etc., the demand for mining and analyzing source code repositories at scale has significantly increased. However, analyzing source code at scale remains expensive to the extent that data-driven solutions to certain SE problems are beyond our reach today. Extant techniques have focussed on leveraging distributed computing to solve this problem, but with a concomitant increase in computational resource needs. This work proposes a technique that reduces the amount of computation performed by the ultra-large-scale source code mining task. Our key idea is to analyze the mining task to identify and remove the irrelevant portions of the source code, prior to running the mining task. We show a realization of our insight for mining and analyzing massive collections of control flow graphs of source codes. Our evaluation using 16 classical control-/data-flow analyses that are typical components of mining tasks and 7 Million CFGs shows that our technique can achieve on average 40% reduction in the task computation time. Our case studies demonstrates the applicability of our technique to massive scale source code mining tasks.

Index Terms—Source code analysis, Mining Software Repositories, Ultra-large-scale Mining, Data-driven Software Engineering.

1 INTRODUCTION
Recently there has been significant interest and success in analyzing large corpora of source code repositories to solve a broad range of software engineering problems including but not limited to defect prediction [1], discovering programming patterns [2], [3], suggesting bug fixes [4], [5], specification inference [6], [7], etc. The approaches that perform source code mining and analysis at massive scale can be expensive. For example, a single exploratory run to mine API preconditions [7] of 3168 Java API methods using a dataset that contains 88,066,029 methods can take 8 hours and 40 mins on a server-class CPU, excluding the time for normalizing, clustering and ranking the preconditions. Often multiple exploratory runs are needed before settling on a final result. The time taken by experimental runs becomes a real hurdle to trying out new ideas.

Fortunately, extant work has focused on leveraging distributed computing techniques to speedup ultra-large-scale source code mining [8], [9], [10], but with a concomitant increase in computational resource needs. As a result, much larger scale experiments have been attempted that perform mining over abstract syntax tree (AST), e.g. [11]. While mining over AST is promising, many SE use-cases seek richer input that further increases the necessary computational costs, e.g., the precondition mining analyzes the control flow graph (CFG). While it is certainly feasible to improve the capabilities of the underlying infrastructure by adding many more CPUs, nonprofit infrastructures e.g., Boa [8] are limited by their resources and commercial clouds can be exorbitantly costly for such tasks.

In this work, we propose a complementary technique that further accelerates ultra-large-scale mining tasks without demanding more computational resources. Given the source code that implements the logic of the mining task, our technique identifies and removes the irrelevant parts of the source code inputs (which can be in millions), prior to running the mining task on the inputs.

Unlike previous ideas to reduce the input source code prior to analyzing it further, most notably program slicing [12], which reduces the input by eliminating portions that are not related to a variable or a statement, accelerating ultra-large-scale source code analysis poses unique challenges because the parts of the source code to be removed is specific to the analysis task that the user wants to perform, the reduction should be sound, in that the analysis results before and after reduction should match, identifying and removing the irrelevant parts must be done efficiently.

Our key insight is that, often a mining task exposes the parts of the source code that are relevant for it. Here, by relevancy we mean, the source code parts that contributes to the mining task output. For instance, if a mining task is about understanding how an API method is being used in the open source projects, the relevant source code parts are the API method invocations. We believe the parts of the source code that are relevant for a mining task can be inferred by analyzing the code of the mining task.

Source code mining can be performed on the source code text or the intermediate representations like abstract syntax trees (ASTs), control flow graphs (CFGs), etc. We apply our insights to mining and analyzing the control flow graphs (CFGs). Given a mining task that is required to be run on a large collection of CFGs, we first perform a static analysis of the mining task to extract a set of rules that can help to identify the relevant nodes in the CFGs. Using the rules, we perform a light-weight pre-analysis that identifies and annotates the relevant nodes in the CFGs. We then perform a sound reduction of the CFGs to remove the irrelevant nodes. Finally, we run the task on the compacted CFGs. Running
the task on the compacted CFGs is guaranteed to produce the same result as running the task on the original CFGs, but without concomitant computational cost.

We have evaluated our acceleration technique using a set of classical iterative control- and data-flow analyses that can be used in the source code mining tasks, for instance dominator and reaching definition analyses are used in the API Precondition mining [7], reaching definition analysis is used in the API usage sequence mining [6], etc. We also present four case studies using concrete source code mining tasks. Both the analyses used in the evaluation and the mining tasks used in the case studies are expressed using Boa (a domain specific language for ultra-large-scale source code mining) [8] and we have used several Boa datasets that contains hundreds to millions of CFGs. On average, we achieved 40% reduction in the task computation time. The reduction depends both on the complexity of the mining task and the percentage of the relevant/irrelevant parts in the input source code.

Contributions. This paper makes the following contributions:

- We introduce a technique to identify the relevant parts of the input for a mining task by analyzing the mining task.
- We show that by performing a light-weight pre-analysis that identifies the relevant parts of the input and soundly prunes the irrelevant parts, greatly reduces the size of the input for the mining task, hence accelerates the mining task, without compromising the accuracy of the results.
- We have implemented our technique in the Boa domain-specific language and infrastructure for ultra-large-scale mining [8].
- Our evaluation shows that for few mining tasks, our technique can reduce the task computation time by over 90%, however on average 40% reduction is seen across a collection of 16 analyses.

2 Overview
An overview of approach is shown in Figure 1. Given an analysis that is expressed as traversals over a CFG and an input CFG, instead of running the analysis directly on the input CFG, we perform a light-weight pre-analysis that identifies the analysis relevant nodes and prunes the analysis irrelevant nodes to obtain a reduced CFG. The pre-analysis stage is helped by a static analysis that provides a set of rules to identify the analysis relevant nodes. The rules contain expressions that when evaluated returns a boolean to identify the analysis relevant nodes. For example, consider the following rule: \((\text{node.expression} == \text{METHODCALL}) \&\& \text{node.expression.method} == \text{"substring"})\). This rule when evaluated on a CFG node, returns true, if the CFG node contains an expression that has a method call to substring API method. Inputs to the pre-analysis stage are: a CFG and a set of rules computed by our static analysis. The pre-analysis stage contains a traversal of the input CFG that identifies and annotates the analysis relevant nodes, and a sound reduction phase that prunes the analysis irrelevant nodes.

Output of the pre-analysis stage is a reduced CFG. We run the analysis on the reduced CFG to produce the output.

Fig. 1. An overview of our approach.

**API Precondition Mining.** To illustrate, consider API Precondition Mining. API preconditions of an API method are the conditions that must be satisfied before calling the API method. Nguyen et al. [7] showed that API preconditions can be inferred by looking at the guard conditions at the API method call sites. Input to their analysis is a set of API methods whose preconditions needs to be computed and a large collection of client methods that calls the API methods. The idea is to analyze the client methods to collect preconditions. Their technique first builds the control flow graphs (CFGs) of the client methods and performs a traversal over the CFGs to identify nodes that calls the API methods. If there are such nodes in the client methods, the mining task performs a dominator analysis to collect nodes on which the API method call nodes are control dependent. The analysis then extracts the predicate expressions from the control dependent nodes, normalizes the expressions in terms of the receivers and the arguments of the API method calls, and outputs the normalized predicate expressions as preconditions.

```java
public void body(String namespace, String name, String text) throws Exception {
    3 String namespaceuri = null;
    4 String localpart = text;
    5 int colon = text.indexOf(":");
    6 if (colon >= 0) {
        7 String prefix = text.substring(0,colon);
        8 namespaceuri = digester.findNamespaceURI(prefix);
        9 localpart = text.substring(colon+1);
    }
    11 ContextHandler contextHandler = (ContextHandler)digester.peek();
    12 contextHandler.addSoapHeaders(localpart,namespaceuri);
}
```

Fig. 2. Code snippet from Apache Tomcat GitHub project.

Consider that we want to mine the API preconditions of substring (int,int) API method using a client method shown in Figure 2. The API precondition mining first builds a CFG of the client method as shown in Figure 3. The node numbers in the CFG corresponds to the line numbers in the code shown in Figure 2. When API precondition mining is run on the CFG of the client method, the analysis visits every node in the CFG and identifies nodes that contains substring (int,int) API method call. In the CFG shown in Figure 3, node 7 contains substring(int,int) API method call. The technique
performs a dominator analysis to determine the nodes on which the API method call is control dependent. In our example, the substring(int,int) call in node 7 is control dependent on nodes 3, 4, 5, and 6. Out of these, the analysis is interested in only the nodes that provides predicate expressions. In this example, only node 6 provides a predicate expression colon \( \geq 0 \). The technique finally returns a normalized predicate expression as precondition.

Figure 3 demonstrates running this task on the CFG of the client method shown in Figure 2. The mining task traverses the CFG multiple times, however the nodes that are of interest to the analysis are only two kinds of nodes: the nodes that contains substring(int,int) API method call and the conditional nodes that provides predicate expressions. All other nodes are irrelevant for the API precondition mining analysis, hence can be removed. A reduced CFG that contains only the nodes that are relevant for the mining task is shown in Figure 3.

Running the analysis on an input program and collecting the execution trace can reveal the parts of the program that are relevant for the analysis. However, the challenge is to identify the analysis relevant parts of the program without running the analysis. For doing that, we perform a static analysis of the analysis code to extract the information about the parts of the program that are relevant for the analysis.

An analysis over a CFG can be described as traversing the nodes in the CFG and executing a block of code. An analysis may require multiple traversals over the CFGs. Our static analysis analyzes each traversal and enumerates acyclic paths in the traversals. There may exists several paths in the traversal, however the paths that produces output and have path conditions on the CFG node are selected. Our idea is that, if any of the selected path is taken at runtime while visiting a CFG node, it means that the CFG node is relevant for the analysis. We know that, a path is taken only when the path conditions are true, hence we formulate a rule for every selected path as a conjunction of the path conditions.

\[
1 \text{ traversal1} \\
2 \quad \text{predicateExprAtNodes: A map containing predicate expression at each node} \\
3 \quad \text{apiCallNodeIds: A list of node ids that contains API method call} \\
4 \quad \text{For each node in the CFG} \\
5 \quad \qquad \text{if node is a branch node} \\
6 \quad \qquad \quad \text{predicate := getPredicateExpr(node)} \\
7 \quad \qquad \quad \text{add <node.id, predicate> to predicateExprAtNodes} \\
8 \quad \quad \text{if node has substring(int,int) api call} \\
9 \quad \quad \quad \text{add node.id to apiCallNodeIds} \\
\]

\[
1 \text{ traversal2} \\
2 \quad \text{dominators: A list of dominator node ids for each node} \\
3 \quad \text{For each node in the CFG} \\
4 \quad \quad \text{compute dominators and add to dominators} \\
\]

\[
1 \text{ traversal3} \\
2 \quad \text{predicates: A list of predicates} \\
3 \quad \text{For each node in the CFG} \\
4 \quad \quad \text{if apiCallNodeIds contains node.id} \\
5 \quad \quad \quad \text{dominators := getDominator(node)} \\
6 \quad \quad \quad \text{for each domid in dominators} \\
7 \quad \quad \quad \quad \text{if predicateExprAtNodes contains domid} \\
8 \quad \quad \quad \quad \quad \text{predicate := getPredicateExpressionAt(domid)} \\
9 \quad \quad \quad \quad \quad \text{add predicate to predicates} \\
\]

To illustrate consider the pseudo code of the API precondition mining shown in Figure 4. The API precondition mining performs three traversals over the CFG of every client method that calls an API method. In traversal1, nodes in the CFG are visited to collect the predicate expressions at nodes and identify nodes that call substring(int,int) API method (lines 5-9 in traversal1). The traversal2 performs a dominator analysis to collect the dominators of every CFG node. In traversal3, nodes in the CFG are visited again to collect the predicate expressions of the dominating nodes of the nodes that calls substring(int,int) API method. At the end of traversal3, predicate expressions of the substring(int,int) API method call are produced as output.

Our static analysis enumerates all acyclic paths in each of the three traversals and selects a subset of the paths that satisfies two conditions: the path produces the output and the path has path conditions on the CFG node. In the three traversal shown in Figure 4, only traversal1 produces such paths, because traversal2 and traversal3 do not have paths with path conditions on the CFG nodes (although they contain paths that produces output). The body of the traversal3 contains four acyclic paths as described in Figure 5. Of these four paths, Path 0 is not selected because it does not produce any output. The other three paths: Path 1, Path 2, and Path 3 produces output and have path conditions on the CFG node. For instance, Path 1 has path conditions \((\text{node is a branch node}) \land \lnot (\text{node has substring(int,int) API method call})\) and it produces an output that contains the predicate expression of the branch node. Similarly other two paths are selected. Each of the three paths yields three rules as the output of our static analysis as shown in Figure 5. These rules...
3 Approach Details

Figure 1 shows an overview of our approach. To recap, the main three components of our approach are: i) a static analysis to extract rules, ii) a pre-analysis traversal to identify the analysis relevant nodes, and iii) a reduction to remove the analysis irrelevant nodes. We now describe each of these three components in detail.

We assume the following formulation for a source code mining task: a source code mining task may contain a set of analysis. A source code analysis such as control flow analysis, data flow analysis, etc., can be expressed as one or more traversals over the control flow graphs (CFGs). A traversal visits every node in the CFG and executes a block of code, often known as an analysis function. An analysis function takes a CFG node as input and produces an output for that node (aka analysis fact). When an analysis function is executed on a CFG node, certain path in the analysis function is taken and this path indicates whether the node is relevant for the analysis or not. We know that, a program path can only be taken if the path conditions along the path are satisfied [13]. Our insight is that, the path conditions can be evaluated in the pre-analysis traversal to identify the nodes in the CFG that are relevant for the analysis. Upon identifying the relevant nodes, the pre-analysis stage performs a sound reduction to prune the unmarked nodes (irrelevant nodes) to produce a reduced CFG. Finally, the analysis is run on the reduced CFG to produce the output.

3.1 Extracting Rules to Infer Analysis Relevant Nodes

Given a set of analysis functions, and input/output variables of the analysis functions, Algorithm 1 computes a rules set that contains path conditions extracted from the paths in the analysis function. For each path, Algorithm 1 visits the nodes in the path and checks if the node is a branch node. For the branch node, it gets the predicate expression contained in the node using an auxiliary function getPredicate(line 7). Algorithm 1 then performs a static analysis to extract this information.

The key idea of our static analysis is to enumerate all acyclic paths of the analysis functions and select a subset of paths based on two conditions: the path has path conditions on the CFG node and the path produces some output. Every selected path produces a rule that is the conjunction of the path conditions. The set of rules when evaluated on CFG nodes helps to identify the relevant nodes for the analysis. We now provide a set of definitions and an algorithm to construct the rules set.

**Definition 1.** A Control Flow Graph (CFG) of a program is defined as $G = (N, E, \top, \bot)$, where $G$ is a directed graph with a set of nodes $N$ representing program statements and a set of edges $E$ representing the control flow between statements. $\top$ and $\bot$ denote the entry and exit nodes of the CFG.

We use the notation $G_A = (N_A, E_A, \top_A, \bot_A)$ to represent the CFG of the analysis function, and $G = (N, E, \top, \bot)$ to represent the CFG of the method that is input to an analysis. A (control flow) path $\pi$ of $G_A$ is a finite sequence $(n_1, n_2, \ldots, n_k)$ of nodes, such that $n_1, n_2, \ldots, n_k \in N_A$ and for any $1 \leq i < k$, $(n_i, n_{i+1}) \in E_A$, where $k \geq 1$, $n_1 = \top_A$ and $n_k = \bot_A$.

A set of paths, $\Pi = \{\pi_0, \pi_1, \ldots\}$ is a set of acyclic paths in the control flow graph $G_A$ of the analysis function. An acyclic path contains nodes that appear exactly once except the loop header node that may appear twice.

**Algorithm 1: Extract rules from an analysis function**

**Input:** Set of paths $\Pi$, CFGNode $i$, Output $ov$  
**Output:** Rules set $R$

1. $R \leftarrow \{\}$
2. foreach $\pi := (n_1, n_2, \ldots, n_k) \in \Pi$ do
3.   $pc \leftarrow true$;
4.   $hasOutputExpr \leftarrow false$;
5.   foreach $n_i \in \pi$ do
6.     if $n_i$ is a branch node then
7.       $pe \leftarrow getPredicate(n_i)$;
8.       $\alpha \leftarrow getAliasAt(iw, n_i)$;
9.       if $getVariables(pe) \cap \alpha \neq \phi$ then
10.      if $n_{i+1}$ is a true successor then
11.        $pc \leftarrow pe \land pe$;
12.      else
13.        $pc \leftarrow pe \land \neg pe$;
14.     else
15.        $\beta \leftarrow getAliasAt(ov, n_i)$;
16.        if $\beta \cap getVariables(n_i) \neq \phi$ then
17.          $hasOutputExpr \leftarrow true$;
18.     if $hasOutputExpr$ is true then
19.       $R \leftarrow R \cup pe$;
20.   end
21. end
22. return $R$;

1. CFGs with multiple exit nodes are converted to structured CFGs by adding a dummy exit node to which all exit nodes are connected.
fetches a list of aliases of the input variable to check if the predicate expression contains the input variable or its aliases (lines 8-9). The idea is to keep only the predicate expressions of the input variable iv. If we decide to keep the predicate expression, we determine whether to add the predicate expression or its negation based on the branch (true or false branch) that the successor node in the path belongs to. The path condition is the "logical and" of all predicate expressions in the path (lines 10-13). If the current visited node is not a branch node, then we get the aliases of the output variable ov and check if the node contains the output variable or its aliases (lines 15-17). The idea here is to keep the path conditions of only those paths that contributes to the output. At the end of visiting all nodes in the path, the computed path condition is added to the rule set R, if the current path contributes to the output (lines 18-19). We use the rule set R computed by Algorithm 1 to produce an annotated control flow graph (ACFG) in the pre-analysis traversal (§3.2).

### 3.1.1 Soundness

The soundness of our static analysis (Algorithm 1) concerns the capability of the analysis to capture all analysis relevant nodes. Missing the analysis relevant nodes may lead to the removal of such nodes which in turn leads to invalid analysis output. Hence, it is important that our static analysis extracts rules that soundly captures all analysis relevant nodes. Using the soundness arguments presented below, we argue that the rules collected by Algorithm 1 are sufficient to identify all analysis relevant nodes. Let us first define the analysis relevant nodes.

**Definition 2.** A node is relevant for an analysis, if it takes a path in the analysis that produces some output. If $\Pi$ is the set of all analysis paths, $\Pi_o \subseteq \Pi$ is the set of all paths such that $\forall \pi := (n_1, n_2, \ldots, n_k) \in \Pi_o$, $\exists n_i$ that produces some output.

For Algorithm 1 to ensure that all analysis relevant nodes will be collected later in the pre-analysis stage, it must ensure that all paths that produces some output are considered.

**Lemma 1.** All paths in the analysis that produces some output ($\Pi_o$) are considered in Algorithm 1.

**Proof sketch.** Algorithm 1 iterates through every path and checks if there exists a statement/expression that writes to the output variable ov to determine if the path should be considered. Provided that getAliasesAt is a sound algorithm [14], we can see that any path that contains statements/expressions that writes to the output variable or its aliases are considered.

We know that, a path is taken if the path conditions along the path are satisfied [13]. So, to ensure that all paths that produces some output are considered, Algorithm 1 must ensure that all path conditions along these paths are collected.

**Lemma 2.** The path conditions along all paths that produce some output ($\Pi_o$) are included in $R$.

2. Before starting the rules extraction process, we first perform an alias analysis to determine aliases of input and output variables using a conservative linear time type-based alias analysis [14].

**Proof sketch.** Given a set of paths that produce output ($\Pi_o$), the input variable iv, a sound algorithm getAliasesAt, the Algorithm 1 extracts all path conditions that involve the input variable iv or its aliases and added to the rules set $R$. We argue that these conditions are sufficient to include all paths that produce output. There are three cases to consider:

- Case 1. When no path contains path conditions (sequential code), then true is added as a rule to ensure that all paths are considered.
- Case 2. When there exists no path that contains conditions on the input variable iv or its aliases, or in other words, no path condition could be added to set $R$, true is added to ensure that all paths are considered.
- Case 3. The cases where there exists some path conditions, but they do not involve the input variable iv or its aliases, true is added to the rules set $R$ to ensure that such paths are included.

In all other cases, the path conditions that involves the input variable iv or its aliases are added to the rules set $R$. We can see that, we do not miss any path that generates output and we collect either the path conditions on the input variable iv or true. Since we do not miss any path that generates output, the relevant nodes which takes the paths that generates output will also not be missed. This is presented as our soundness theorem next.

**Theorem 3.** (Soundness). If $N_R \subseteq N$ is a set of all relevant nodes, $\forall n \in N_R, \exists r \in R$, such that evaluates($r; n$) is true, where the auxiliary function evaluates given a rule $r$ (which is a predicate) and a CFG node, checks the satisfiability to return true or false.

**Proof sketch.** By Lemma 1 and Lemma 2, the theorem holds.

### 3.1.2 Time Complexity

The time complexity of Algorithm 1 is $O(p \times n)$, where $p$ is the number of acyclic paths in the CFG of the analysis function (can grow exponentially, but finite) and $n$ is the number of nodes in the CFG of the analysis function (the number of nodes can be considered nearly equal to the number of program statements). Prior to Algorithm 1 there are two key steps: computing the acyclic paths and computing the alias information. Our acyclic path enumeration step performs a DFS traversal of the CFG that has $O(n + e)$ time complexity in terms of number of nodes $n$ and number of edges $e$. The alias analysis used in our approach is a type-based alias analysis [14] that has a linear time complexity in terms of the number of program statements.

### 3.2 Annotated Control Flow Graph

In §3.1 we described our static analysis to extract rules. The output of our static analysis is a set of rules that

3. It is not possible that some paths contain path conditions and other don’t. It is either all paths contain path conditions or none because if a path condition along one path exists then the negation of that path condition also exists along other paths.
Definition 5. A Reduced Control Flow Graph (RCFG) of an ACFG \( G' = (N, E, N_R, \top, \bot) \) is a pruned ACFG with analysis irrelevant nodes pruned. A RCFG is defined as \( G'' = (N', E', \top', \bot') \), where \( G'' \) is a directed graph with a set of nodes \( N' \subseteq N \) representing program statement and a set of edges \( E' \) representing the control flow between statements. \( \top \) and \( \bot \) are the entry and exit nodes. The edges \( E - E' \) are the removed edges and \( E' - E \) are the newly created edges.

Algorithm 3: Build RCFG

\[ G'' \leftarrow G'; \]
\[ \text{foreach node } n \text{ in } G'' \text{ do} \]
\[ \text{if } n \notin N_R \text{ then} \]
\[ \text{Adjust}(G'', n); \]
\[ \text{remove } n \text{ from } G''; \]
\[ \text{foreach node } n \text{ in } G'' \text{ do} \]
\[ \text{if } n \text{ is a branch node then} \]
\[ \text{if } \text{SuccOf}(n) \text{ contains } n \text{ then} \]
\[ \text{remove } n \text{ from } \text{SuccOf}(n); \]
\[ \text{if } n \text{ has only one successor then} \]
\[ \text{Adjust}(G'', n); \]
\[ \text{remove } n \text{ from } G''; \]
\[ \text{return } G''; \]

Definition 6. An Annotated Control Flow Graph (ACFG) of a CFG \( G = (N, E, \top, \bot) \) is an ACFG \( G' = (N, E, N_R, \top, \bot) \) with a set of nodes \( N_R \subseteq N \) computed using Algorithm 2.

Definition 5. Given an ACFG \( G' = (N, E, N_R, \top, \bot) \), a node \( n \in N \) is an analysis relevant node if:
- \( n \) is a \( \top \) or a \( \bot \) node,
- \( n \) is also in \( N_R \), but not a branch node,
- \( n \) is a branch node with at least one branch that has an analysis relevant node.

3.3 Reduced Control Flow Graph

Using the annotated control flow graph (ACFG) that contains a set \( N_R \) of probable analysis relevant nodes, we perform a sound reduction that refines the set \( N_R \) and also removes the analysis irrelevant nodes to create a reduced or compacted CFG called a reduced control flow graph (RCFG). An RCFG is a pruned CFG that contains only the analysis relevant nodes. An RCFG is constructed by performing a reduction on the ACFG.

Algorithm 4: A procedure to adjust predecessors and successors of a node being removed

\[ \text{Procedure Adjust} \ (\text{CFG } G, \text{ CFGNode } n) \]
\[ \text{foreach predecessor } p \text{ of } n \text{ in } G \text{ do} \]
\[ \text{remove } n \text{ from } \text{SuccOf}(p); \]
\[ \text{foreach successor } s \text{ of } n \text{ in } G \text{ do} \]
\[ \text{remove } n \text{ from } \text{PredsOf}(s); \]
\[ \text{add } p \text{ to } \text{PredsOf}(s); \]
\[ \text{add } s \text{ to } \text{SuccOf}(p); \]

3.4 ACFG To RCFG Reduction

Algorithm 3 describes the reduction from ACFG to RCFG. The algorithm visits the nodes in the ACFG and checks if the node exists in \( N_R \). The nodes that does not exists in \( N_R \) are pruned (lines 2-5). Before removing an analysis irrelevant node, new edges are created between the predecessors and the successors of the node that is being removed (line 4 and Algorithm 4). After removing all analysis irrelevant nodes, we pass through the RCFG and remove irrelevant branch nodes (lines 6-11). Irrelevant branch nodes are those branch nodes that have only one successor in the RCFG (this node is
no longer a valid branch node). Note that, our definition of analysis relevant nodes (Definition 5) includes branch nodes with at least one branch that has an analysis relevant node.

For flow-insensitive analysis, the analysis results depends only on the results produced at nodes. For flow-sensitive analysis, the analysis results depends on the results produced at nodes, and the flow of results between nodes.

It is easy to see that, for flow-insensitive analysis, the analysis results of RCFG and CFG should match, because all the nodes that produce results are retained in the RCFG.

For flow-sensitive analysis, the result producing nodes in RCFG and CFG are same. For the flow of results between nodes in RCFG and CFG to be same, the flow between nodes in the CFG should be retained for the corresponding nodes in the RCFG and no new flows should be created.

Definition 7. Given any two nodes \( n_1 \) and \( n_2 \) of a CFG \( G \), the analysis results can flow from \( n_1 \) to \( n_2 \), iff there exists a path \( n_1 \to^* n_2 \). This flow is represented as \( n_1 \to^* n_2 \).

Lemma 3. The flow between analysis relevant nodes in the CFG should be retained for the corresponding nodes in the RCFG. That is, for any two analysis relevant nodes \( n_1 \) and \( n_2 \) in the CFG \( G \), if \( n_1 \to^* n_2 \) exists in \( G \), then \( n_1 \to^* n_2 \) should also exists in RCFG \( G'' \).

Proof sketch. For ensuring flows in the CFG is retained in the RCFG, every path between any two analysis relevant nodes in the CFG should have a corresponding path between those nodes in the RCFG. This is ensured in our reduction algorithm (Algorithm 3), where for removing a node, an edge from each predecessors to each successors is established (lines 4 and 10). If there exists a flow \( n_1 \to^* n_2 \) for a path \( n_1 \to n_k \to n_2 \) via an intermediate node \( n_k \), the Algorithm 3, while removing \( n_k \), establishes a path \( n_1 \to n_2 \) by creating a new edge \((n_1, n_2)\), and hence the flow \( n_1 \to^* n_2 \) is also retained.

Lemma 4. No new flows should be created between nodes in the RCFG that does not exists between the corresponding nodes in the CFG. For any two analysis relevant nodes \( n_1 \) and \( n_2 \) in the CFG \( G \), if \( n_1 \to^* n_2 \) does not exists in \( G \), then \( n_1 \to^* n_2 \) should not exists in RCFG \( G'' \).

Proof sketch. For ensuring no new flows are created, every path between any two nodes in the RCFG should have a corresponding path between those nodes in the CFG. This is ensured in our reduction algorithm (Algorithm 3), where for removing a node, an edge from each predecessors to each successors is established, iff there exists a path from the predecessor to the successor in the CFG. For any two analysis relevant nodes \( n_1 \) and \( n_2 \) in the CFG \( G \), Algorithm 3 ensures that, while removing a node \( n_{k_2} \) in a path \( n_1 \to n_k \to n_2 \), a new edge between \( n_1 \to n_2 \) is created in \( G'' \), if there exists a path \( n_1 \to n_2 \) in \( G \). Hence Algorithm 3 guarantees that no new paths are created, and hence no new flows are created.

Theorem 8. For flow-sensitive analysis, the analysis results for RCFG and CFG are same.

Proof sketch. For flow-sensitive analysis, the fact that the result producing nodes in RCFG and CFG are same, and by Lemma 3 and Lemma 4, it follows that analysis results for RCFG and CFG are same.
3.6 Efficiency of Reduction

Our reduction algorithm has linear time complexity in terms of the CFG size. The reduction has two pass over the CFG nodes, where in the first pass the analysis irrelevant nodes are pruned (lines 2-5 in Algorithm 3) and in the second pass the irrelevant branch nodes are removed (lines 6-11).

4 Empirical Evaluation

We evaluated our acceleration technique using 16 source code analyses that mainly contains foundational control and data flow analysis as listed in Table 1. Analyses are written using Boa [8], a domain specific language (DSL) for ultra-large-scale mining, and we have used two Boa datasets: DaCapo and SourceForge for running the analyses. The DaCapo dataset contains 304,468 control flow graphs extracted from the 10 GitHub Java projects [15], and SourceForge dataset contains over 7 million control flow graphs extracted from the 7,029 SourceForge Java projects 7. We compare our reduced control flow graph (RCFG) approach against a Baseline, where, in RCFG, the analysis is run on the reduced control flow graph, and in Baseline, the analysis is run on the control flow graph. We measure the analysis time for Baseline and RCFG approaches using the methodology proposed by Georges et al. [16], where analysis times are averaged over three runs, when the variability across these measurements is minimal (under 2%). Our experiments were run on a machine with 24 GB of memory and 24-cores, running on Linux 3.5.6-1.fc17 kernel. The RCFG approach time includes the time for pre-analysis (identifying and annotating relevant nodes) and the time for reduction (removing irrelevant nodes) along with the actual analysis time. Whereas, the Baseline time only includes the actual analysis time.

4.1 Reduction In Analysis Time

We measure reduction in the analysis time of RCFG over Baseline. The results of our measurement is shown in Table 1. For instance, the Available Expressions (AE) analysis when run on DaCapo dataset that contains 304,468 CFGs took 13.31s in the Baseline approach and 6.37s in the RCFG approach. The reduction in the analysis time for AE is 6.93s and the percentage reduction in the analysis time is 52.09%.

Table 1 also shows the minimum, maximum, average, and median values of both reduction and % reduction in the analysis time. From these values it can be seen that, on average (across 16 analyses) our approach was able to save 5s on the DaCapo dataset and 104s on the SourceForge dataset. In terms of the percentage reduction, on average, 41% reduction is seen for the DaCapo dataset and 42% reduction is seen for the SourceForge dataset. Our approach was able to obtain the maximum reduction for the Precondition Mining (PM) analysis, were on the DaCapo dataset, 39s (95%) were saved and on the SourceForge dataset, 878s (96%) were saved. Across DaCapo and SourceForge datasets, for 11 out of 16 analysis, our approach was able to obtain a substantial reduction in the analysis time. For 5 analysis (highlighted in gray), the reduction is either negative or negligible. We first discuss the favorable cases and then provide insights into unfavorable cases.

Reduction in the analysis time stems from the reduction in the graph size of RCFG over CFG, hence we measured the graph size reduction in terms of Nodes, Edges, Branches, and Loops for understanding the analysis time reductions. We accumulated these metrics over all the graphs in the datasets. The results of the measurement is shown in Table 1 under Graph Size (% Reduction) column. The reduction in graph size is highly correlated to the reduction in analysis time. Higher the reduction in graph size, higher will be the reduction in analysis time. For instance, consider the Precondition Mining (PM) analysis that had 95% and 96% reduction in the analysis time on DaCapo and SourceForge datasets respectively. For PM, the reduction in the graph size in terms of Nodes, Edges, Branches, Loops, were 62.42%, 70.27%, 57.9%, 56.27% for DaCapo, and 63.98%, 71.9%, 59.25%, 56.91% for SourceForge dataset. To summarize the favorable results, it can be seen that for 11 of 16 analysis, our technique was able to reduce the analysis time substantially (on average over 60% reduction for 11 analyses, over 40% reduction over all analyses). This reduction can be explained using the reduction in graph size in terms of Nodes, Edges, Branches, and Loops. Further, Table 2 lists the relevant parts of the code for various analysis to give more insights into the reduction. The analysis that contains common statements as relevant parts sees less reductions. For instance, CP has variable definitions and variable accesses as relevant statements, which are very common in majority of the source code, hence sees very less reductions. Whereas, PM has String.substring API method calls and predicate expressions as relevant statements, which are not very common in majority of the source code, hence sees very high reductions.

For 5 analysis, the reduction in analysis time is either negative or negligible. The reduction in graph sizes for these 5 analysis are also low. These analysis are: constant propagation (CP), copy propagation (CP'), dead code (DC), live variables (LV), and reaching definitions (RD). One thing to notice is that, for all these five analyses, the set of relevant statements are same: variable definitions and variable accesses, which are frequent in any source code. Hence, for these analysis the graph size of the RCFG is similar to the CFG, and our technique could not reduce the graph size much. Since the graph size for RCFG and CFG are similar, their analysis times will also be similar (not much reduction in the analysis time). For some analysis, the RCFG approach time exceeds the Baseline approach time due to the additional overheads that RCFG approach has for annotating and reducing the CFG to produce RCFG. From these unfavorable results we can conclude that for analysis for which the reduction in graph size is not substantial, the RCFG may incur overhead leading to larger analysis time than Baseline. However, the overhead is not substantial. For instance, for DaCapo, CP: -4.75%, CP': 0.005%, DC: -13.64%, LV: -7.7%, and RD: -6.87%. For a larger dataset, such as SourceForge, the overheads are further small: CP: -0.16%, CP': 0.08%, DC: 1.4%, LV: 0.62%, RD: 2.32%. This indicates that, the analysis that are unfavorable to the RCFG approach, do not incur substantial overheads.

Next, we show the boxplots representing % reduction in
TABLE 1
Reduction in analysis time and reduction in graph size for DaCapo and SourceForge datasets over 16 analysis. The column CFG provides the analysis time in baseline approach and the column RCFG provides the analysis time in our approach. RCFG analysis time includes the annotation and reduction overheads. Column R provides the reduction in the analysis time and % R provides the percentage reduction in the analysis time. Under Graph Size (% Reduction), the columns N, E, B, L represents nodes, edges, branches, and loops. The table also provides the minimum, maximum, average, and median for both reduction (R) and percentage reduction (%R) in the analysis time.

<table>
<thead>
<tr>
<th>Analysis</th>
<th>DaCapo</th>
<th>SourceForge</th>
<th>Analysis Time (seconds)</th>
<th>DaCapo</th>
<th>SourceForge</th>
<th>Graph Size (%Reduction)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CFG</td>
<td>RCFG</td>
<td>R</td>
<td>%R</td>
<td>CFG</td>
<td>RCFG</td>
</tr>
<tr>
<td>1</td>
<td>Available Expressions (AE)</td>
<td>13.21</td>
<td>6.29</td>
<td>6.93</td>
<td>52.59</td>
<td>139.53</td>
</tr>
<tr>
<td>2</td>
<td>Common Sub. Elimination (CSE)</td>
<td>13.96</td>
<td>6.25</td>
<td>7.22</td>
<td>55.26</td>
<td>157.90</td>
</tr>
<tr>
<td>3</td>
<td>Constant Propagation (CP)</td>
<td>9.84</td>
<td>10.31</td>
<td>0.47</td>
<td>4.75</td>
<td>315.59</td>
</tr>
<tr>
<td>4</td>
<td>Lopp Propagation (CP)</td>
<td>13.29</td>
<td>13.29</td>
<td>0.00</td>
<td>0.00</td>
<td>380.33</td>
</tr>
<tr>
<td>5</td>
<td>Dead Code (DC)</td>
<td>13.66</td>
<td>15.52</td>
<td>-1.86</td>
<td>-13.64</td>
<td>501.96</td>
</tr>
<tr>
<td>6</td>
<td>Live Variables (LV)</td>
<td>4.83</td>
<td>5.20</td>
<td>-0.37</td>
<td>-7.70</td>
<td>173.42</td>
</tr>
<tr>
<td>7</td>
<td>Loop Must Not Alias (LMNA)</td>
<td>3.93</td>
<td>4.21</td>
<td>-1.32</td>
<td>-5.82</td>
<td>171.75</td>
</tr>
<tr>
<td>8</td>
<td>Local Must Not Alias (LMNA)</td>
<td>5.80</td>
<td>2.08</td>
<td>3.72</td>
<td>64.18</td>
<td>158.49</td>
</tr>
<tr>
<td>9</td>
<td>Loop invariant (LI)</td>
<td>11.57</td>
<td>14.27</td>
<td>-2.72</td>
<td>-34.63</td>
<td>318.17</td>
</tr>
<tr>
<td>10</td>
<td>Precodination Mining (PM)</td>
<td>41.49</td>
<td>21.25</td>
<td>29.54</td>
<td>94.58</td>
<td>91.28</td>
</tr>
<tr>
<td>12</td>
<td>Resource Leak (RL)</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>93.38</td>
<td>0.56</td>
</tr>
<tr>
<td>13</td>
<td>Safe Syntactic.localization (SS)</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>61.11</td>
<td>0.02</td>
</tr>
<tr>
<td>14</td>
<td>SIant (TA)</td>
<td>0.97</td>
<td>0.35</td>
<td>-0.42</td>
<td>-43.49</td>
<td>20.54</td>
</tr>
<tr>
<td>15</td>
<td>Upsafety (UP)</td>
<td>13.24</td>
<td>5.99</td>
<td>7.25</td>
<td>54.76</td>
<td>139.07</td>
</tr>
<tr>
<td>16</td>
<td>Very Busy Expressions (VBE)</td>
<td>14.15</td>
<td>14.27</td>
<td>-0.37</td>
<td>-6.34</td>
<td>44.95</td>
</tr>
</tbody>
</table>

Table 2 Relevant source code parts for various analyses.

<table>
<thead>
<tr>
<th>AE</th>
<th>CSE</th>
<th>CP</th>
<th>CP'</th>
<th>DC</th>
<th>LV</th>
<th>LMA</th>
<th>LMNA</th>
<th>LI</th>
<th>PM</th>
<th>RD</th>
<th>RL</th>
<th>SS</th>
<th>TA</th>
<th>UP</th>
<th>VBE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Assignment statements with RHS contains variables, but not method call or new expression</td>
<td>Same as AE</td>
<td>2) Variable definitions</td>
<td>Same as CP</td>
<td>2) Variable accesses</td>
<td>Same as CP'</td>
<td>1) Variable definitions of loop variables</td>
<td>Same as LMA</td>
<td>2) Variable accesses of loop variables</td>
<td>3) Loop statements (FOR, WHILE, DO)</td>
<td>1) String.substring API method calls</td>
<td>Contains read, write or close method calls from InputStream</td>
<td>1) Variable definitions with RHS contains Console.readLine or FileInpuStream.read method calls</td>
<td>Same as CP</td>
<td>1) Binop expression that have variable access, but are not method calls</td>
<td>Same as AE</td>
</tr>
</tbody>
</table>

Analysis time and graph size across 16 analysis. The box plot says that, for DaCapo dataset, the % reduction in analysis time is in between -2.37 to 62.26 (first and third quartiles) with median 53.42. For SourceForge dataset, the reduction in analysis time is in between 1.86 to 63.57 with median 49.92. These numbers indicates that, i) for most analysis the reduction in the analysis time is substantial (indicated by the median and third quartile), and ii) for the analyses that does not benefit from the RCFG approach, do not incur too much overhead (indicated by the first quartile).

To summarize, our set of 16 analyses showed great reduction in the analysis times were between several seconds to several minutes (the maximum was 15 minutes), however the percentage reduction in the analysis times were substantial (the maximum was 96%). Though, the reduction seems small, as we show in our case study of several actual ultra-large-scale mining tasks that utilizes several of these foundational analyses, when run on an ultra-large dataset (containing 162 million CFGs, 23 times larger than our SourceForge dataset) can achieve substantial reduction (as much as an hour). Further, the ultra-large-scale mining tasks that we target are often the exploratory analysis which requires running the mining tasks several times, where the results of the mining task are analyzed to revise the mining task and rerunning it, before settling on a final result. The time taken by the experimental runs becomes a real hurdle to trying out new ideas. Moreover, the mining tasks are run as part of the shared infrastructure like Boa [8] with many concurrent users (Boa has more than 800 users), where any time/computation that is saved has considerable impacts, in that, many concurrent users can be supported and the response time (or the user experience of users) can be significantly improved.
In terms of which analyses can benefit from our approach, we can see that the reduction in the analysis time depends both on the complexity of the analysis and the percentage of the program that is relevant for the analysis. For those analyses for which most of the program parts are relevant (or in other words, an input program cannot be reduced to a smaller program), our technique may not be very beneficial. For those analyses for which the relevant program parts are small, our technique can greatly reduce the analysis time. Also, for analyses that have simple complexity, for instance, analysis that perform single traversal (or parses the program once) may not benefit from our approach. Ideal scenarios for our technique to greatly help is when the analysis requires multiple traversals over the program (or program graphs) and the analysis relevant parts are small in the input programs.

4.2 Scalability

In this section, we measure the scalability of Baseline and RCFG approaches over increasing dataset sizes for our 16 analyses. For simulating the increasing dataset sizes, we have divided our 7 million CFGs of SourceForge dataset into 20 buckets, such that each bucket contains equal number of graphs with similar characteristics in terms of graph size, branches, and loops. Using the 20 buckets, we created 20 datasets of increasing sizes (\(D_0\) to \(D_{19}\)), where \(D_t\) contains graphs in \(\text{bucket}_t\) to \(\text{bucket}_t\).

We measure the analysis time of the Baseline and the RCFG approaches for all 16 analyses and plot the result in Figure 8. Our results shows that, as the dataset size increases, for both Baseline and RCFG, the analysis time increases sub-linearly. Our results also shows that, for increasing dataset sizes, RCFG performs better than Baseline for 11 of 16 analyses (where RCFG line is below Baseline line in the charts). For 5 analyses Baseline is better than RCFG. These analyses are the unfavorable candidates that we discussed previously (CP, CP’, DC, LV, and RD).

4.3 Accuracy & Efficiency of Reduction

We evaluate the accuracy of the reduction by comparing the results of the RCFG and the Baseline approaches. We used DaCapo dataset for running the analyses and comparing the results. We found that, for all the analysis, the two results match 100%. This was expected, as RCFG contains all the nodes that produce output, RCFG retains all flows between any two nodes, and RCFG does not introduce new flows.

For evaluating the efficiency of reduction, we measured time for different components of the RCFG approach. The RCFG approach has three components: 1) traversal that annotates analysis relevant nodes, 2) reduction that prunes analysis irrelevant nodes, and 3) the actual analysis. Figure 9 shows the distribution of the RCFG time over these three components for DaCapo and SourceForge datasets over 16 analysis. The data labels provide the numbers for the RCFG time in seconds. From the results shown in Figure 9, it can be seen that, majority of the RCFG time is contributed by the actual analysis and not the overheads. We see, for some analysis, the traversal (that annotates the relevant nodes) contributes more than the actual analysis (RL in DaCapo, SS in SourceForge), however, for all analysis the
reduction time is negligible, when compared to the actual analysis time. Further, we measured the time for each of the three components and aggregated it for all 16 analysis for all the graphs in the DaCapo and SourceForge datasets. For DaCapo, the traversal, reduction, and analysis times were 0.398, 0.047, and 92.268 seconds respectively, and for SourceForge, the traversal, reduction, and analysis times where 1.702, 1.055, 2288.295 seconds respectively. As we can see, the reduction time for both DaCapo and SourceForge datasets is very negligible when compared to the analysis time for all 16 analysis. In summary, analysis results of RCFG and Baseline match 100%, indicating the soundness of the reduction. The negligible time for reduction when compared to actual analysis time, indicates that our reduction is efficient.

4.4 Overhead of Static Analysis

We presented our static analysis in section §3.1 and discussed its time complexity. In this section, we present our measurements of the overhead of the static analysis for all the 16 analyses. Table 3 presents these overheads along with some characteristics of the analyses, such as number of lines of code (Boa program LOC), number of analysis functions (or the CFGs), and number of paths (total number of paths in all the CFGs that are analyzed). Table 3 also presents the total overhead ($T_{total}$) of our static analysis along with the overheads of each of its components: CFG building time ($T_1$), path generation time ($T_2$), alias analysis time ($T_3$), and rules extraction time ($T_4$).

Based on the median value over 16 analyses, the overhead is around 300ms. What this means is that, the compilation time of the analysis program is increased by 300 milliseconds. A majority of this overhead is contributed by the path enumeration phase. We can see the worst case overheads for two analyses: LMA and TA. In both the cases, the overheads are large due to the large amount of time required for path enumeration. These analyses have deeply nested branches and loops as part of their analyses functions which increases the number of paths and the path enumeration time. In summary, as our static analysis explores paths in the analysis, analysis with many paths may incur a non-negligible overhead, however this overhead is an one-time compilation overhead.

4.5 Threats to Validity

A threat to validity is for the applicability of our results. We have studied several source code mining tasks that perform control- and data-flow analysis and showed that significant acceleration can be achieved (on average 40%). However, these results may not be true for mining tasks that have completely different characteristics than the studied subjects. To mitigate this threat, we have included the tasks that have varying complexities in terms of the number of CFG traversals they require and the operations performed by them. We did not had to worry about the representativeness of our dataset that contains CFGs, because the dataset is prepared using the open source code repositories with thousands of projects and millions of methods, which often includes all kinds of complex methods. Further, the amount of reduction that our technique is able to achieve shows significant variations validating our selection of mining tasks. We haven’t considered the mining tasks that requires global analysis such as callgraph analysis or inter-procedural control flow analysis. We plan to investigate them as part of our future work.

5 Case Studies

In this section, we show the applicability of our technique using several concrete source code mining tasks. We run these tasks on a humongous dataset containing 162 million CFGs drawn from the Boa GitHub large dataset. We use the distributed computing infrastructure of Boa to run the mining tasks. We profile and measure the task time and compare

![Image](image-url)
the two approaches, Baseline and RCFG, to measure the acceleration.

5.1 Mining Java API Preconditions

In this case study, we mine API preconditions of all the 3168 Java API methods as discussed in §2. The mining task contains three traversals. First traversal collects the nodes with API method calls and predicate expressions. Second traversal performs a dominator analysis. Third traversal combines the results of the first two traversals to output the predicate expressions of all dominating nodes of the API method call nodes. The Baseline approach took 2 hours, 7 minutes and 40 seconds and the RCFG approach took 1 hour, 6 minutes and 51 seconds to mine 11,934,796 client methods that called Java API methods. Overall, 47.63% reduction in the task computation time. For analyzing the results, we also measured the % graph size reductions in terms of Nodes, Edges, Branches, and Loops. The values were, 41.21, 46.10, 37.91, and 37.57 respectively. These numbers indicate substantial reduction in the graph size of RCFG when compared to CFG. The nodes that are relevant for the task are the nodes that call API methods and the conditional nodes that provide predicate expressions. All other nodes are irrelevant and they do not exist in the RCFGs. One can expect that in the client methods that calls Java API methods, there are significant amount of statements not related API method call or predicate expressions, as we show an example client method in Figure 2. This explains the reduction in the task time.

5.2 Mining Java API Usage Sequences

In this case study, we mine the API usage sequences of all the 3168 Java API methods. An example API usage sequence is Iterator.hasNext() and Iterator.next(). For mining the usage sequences, the mining task traverses the CFGs to identify API method calls. If a CFG contains two or more API method calls, it performs a data-flow analysis to determine the data-dependence between the API method invocations [6], [17]. Finally, the task outputs the API method call sequences that are data-dependent for offline clustering and determining the frequent API call sequences (which APIs are used together often). For this task, the Baseline approach took 1 hour, 33 minutes and 5 seconds and the RCFG approach took 1 hour, 16 minutes and 20 seconds to mine 24,479,901 API usage sequences. These API sequences can be used for clustering and computing the frequently occurring API sequences. Overall, 18% reduction in the task computation time. The nodes that are relevant for the task are: API method call nodes and the nodes that define the variables used by the API method calls. All other nodes are irrelevant. Here, the opportunity for reduction is less, as all the statements that contains variable definitions are relevant along with the API method call statements and the variable definitions are quite common in source codes. The % graph size reduction metrics supports our reasoning, where the values were: (Nodes, Edges, Branches, Loops) = (17.99, 18.32, 11.12, 5.21), on the lower side.

8 The task time excludes the distributed job configuration time and the CFG building times, because these are same for both Baseline and RCFG approaches. However, the RCFG time includes all runtime overheads (annotation and reduction overheads).

5.3 Mining Typically Synchronized Java APIs

In this case study, we mine the typically synchronized Java API method calls to help inform when the API methods are used without synchronization. In other words, the Java API method calls that are protected using the lock primitives. The task first traverses the CFGs to determine if there exists safe synchronization using Java locking primitives (java.util.concurrent.locks). There exists a safe synchronization if all the locks acquired are release along all program paths within the method. In the next traversal, the task identifies all API method calls that are surrounded with safe synchronization and output them to compute the most protected Java APIs. According to our mined results, the top 5 synchronized Java API method calls were:

1) Condition.await()
2) Condition.signalAll()
3) Thread.start()
4) Iterator.next()
5) Iterator.hasNext()

We were surprised to see Thread.start() in the top 5, however manually verifying many of the occurrences indicated that the source code related to the project’s test cases often surround Thread.start() with locks.

For this mining task, nodes that are relevant are: lock() and unlock() API method call nodes, and the Java API method call nodes. For this task, the Baseline approach took 11.1 seconds and the RCFG approach took 8.45 seconds, i.e., 23.72% reduction in the task computation time. The % graph size reduction metrics Nodes, Edges, Branches, and Loops were 32.12, 35.33, 25.21, and 18.46 respectively, supports the reduction in the task time.

5.4 Mining Typically Leaked Java APIs

In this case study, we mine the typically leaked Java APIs. There exists 70 Java APIs for managing the resources, such as InputStream, OutputStream, BufferedReader, etc. A resource can be leaked if it is not closed after its use. The mining task performs a resource leak analysis that captures if a resource used is not closed along all program paths. The task collects all the resources that are used along with program (as analysis facts), propagates them along the program using flow analysis, and checks if any resource is not closed at the program exit point. According to our mined results, the top 5 Java resources that often leaked were:

1) java.io.InputStream
2) java.sql.Connection
3) java.util.logging.Handler
4) java.io.OutputStream
5) java.sql.ResultSet

The nodes that are relevant for this mining task are the resource related API method call nodes. All other nodes are irrelevant. For this task, the Baseline approach took 6 minutes and 30 seconds and the RCFG approach took 6 minutes and 18 seconds, i.e., only 2.97% reduction in the

9 Note that, both lock/unlock and resource leaks may go beyond a single method boundary. Such cases are not considered, as we do not perform an inter-procedural analysis.
task computation time. We expected significant reduction in the task time using RCFGs, however the results were contradictory. For further analysis, we measured the % graph size reduction metrics: Nodes, Edges, Branches, and Loops, whose values were, 42.31, 44.91. 38.81, 37.23 respectively, shows significant reduction only added to our surprise. Further investigation indicated that, although the RCFGs were much smaller than CFGs, the complexity of the mining task was small enough, such that the benefit obtained by running the task on the RCFGs were overshadowed by the overhead of the pre-analysis traversals and the reductions in our approach.

6 Related Works

There has been works that accelerate analysis by performing analysis specific pre-analysis and program compaction [18], [19]. Allen et al. [18] propose a staged points-to analysis framework for scaling points-to analysis to large code bases. They perform static program slicing and compaction to reduce the input program to a smaller program that is semantically equivalent for the points-to queries under consideration. Reduction in the number of variables and allocation sites is the key to acceleration. Smaragdakis et al. [19] propose an optimization technique for flow-insensitive points-to analysis, in which the source program is transformed by a set-based pre-analysis, prior to the value-based points-to analysis that reasons about the flow of points-to facts. The acceleration stems from the reduction of program’s local variables and context-sensitive points-to facts. Analysis of the original and the transformed programs yields the same points-to facts. When compared to these works, our technique of pre-analysis and graph compaction is not specific to an analysis. Given an analysis, we perform a static analysis to extract the information about the relevant parts of the graph, hence our approach can be generally applicable to any analysis that is expressed as graph traversals, however, in this work, we target control and data flow analysis. In our work, the acceleration stems from the reduction in the graph size in terms of nodes, edges, branches, and loops.

The concept of identifying and removing the irrelevant parts has been used in other approaches to improve the efficiency of the techniques [20], [21]. For instance, Wu et al. [20] uses the idea to improve the efficiency of the call trace collection and Ding et al. [21] uses the idea to reduce the number of invocations of the symbolic execution in identifying the infeasible branches in the code. Both Wu et al. and Ding et al. identifies the relevant parts of the input for the task in hand. The task in these approaches is fixed. In Wu et al. the task is call trace collection and in Ding et al. the task is infeasible branch detection using symbolic execution. Whereas, in our technique the task varies and our technique identifies the relevant parts of the input for the user task by analyzing the task.

There has been efforts to scale path sensitive analysis of programs by detecting and eliminating infeasible paths (pruning the paths) before performing the analysis [22], [23]. Our approach goes beyond the path sensitive analysis, where control and data flow analysis that are flow-sensitive and path-insensitive, and explores some paths can also benefit from our approach. Moreover, our technique can automatically filter the analysis relevant parts by performing a static path analysis, whereas their work specifically filters event nodes for a 2-event path sensitive analysis, such as safe synchronization that filters lock and unlock event nodes.

Our work is also similar to the works that accelerate program analysis [24], [25], [26], [27]. Kulkarni et al. [24] proposed a technique to accelerate program analysis in Datalog. Their technique runs an offline analysis on a corpus of training programs and learns analysis facts over shared code. It reuses the learnt facts to accelerate the analysis of other programs that share code with the training corpus. Other works that performs pre-analysis of the library code to accelerate analysis of the programs that make use of the library code exists [25], [26], [27]. When compared to their approach, our approach does not require programs to share code.

Reusing analysis results to accelerate interprocedural analysis by computing partial [28] or complete procedure summaries [29], [30] is also studied. These works computes partial/complete summary of each procedure in a program, and applies each summary to analyze the procedure at each call site. Our technique does not reuse the analysis results, instead it transforms (in a way summarizes) the input graph by keeping only the analysis relevant nodes to accelerate the analysis.

Program slicing technique is used to remove statements irrelevant for the given slicing criteria [12]. Lokuciejewski et al. [31] used program slicing to accelerate static loop analysis. Our technique of reducing the program to contain only the analysis relevant parts can be considered as program slicing, where the slicing criteria would be that nodes that are relevant for the analysis. The difference is that, a program slice is executable, RCFG is not, although, RCFG and CFG produce the same results for a given analysis.

7 Conclusion

Data-driven software engineering demands mining and analyzing source code repositories at massive scale, which can be expensive. The extant techniques have focused on leveraging the distributed computing techniques to solve this problem, but with a concomitant increase in the computational resource needs. This work proposes a complementary technique that reduces the amount of computation performed by the ultra-large-scale source code mining tasks without compromising the accuracy of the results. The key idea is to analyze the mining task to identify and remove the parts of the source code that are irrelevant for the mining task prior to running the mining task. We show a realization of our insights for mining tasks that perform control- and data-flow analysis at massive scale. Our evaluation using 16 classical control- and data-flow analysis showed substantial reduction in the mining task time. Our case studies demonstrated the applicability of our technique to massive scale source code mining tasks.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under grants CCF-08-46059, CCF-11-17937, and CCF-14-23370. The views expressed in this
work are those of the authors and do not necessarily reflect the official policy or position of the NSF. The authors would like to thank the anonymous reviewers of ESEC/FSE 2017 and Dr. Wei Le for constructive comments and suggestions for improving the manuscript.

References


Ganesh Upadhyaya is a doctoral candidate at Iowa State University. His research interests include program analysis, mining software repositories, and concurrent programming. He has published and presented several works at OOPSLA, ICSE, MSR, Modularity, and AGERE. He is a member of the IEEE.

Hridesh Rajan is the Kingland Professor in the Computer Science Department at Iowa State University. His research interests include program analysis, mining software repositories, and concurrent programming. He has been recognized by the US National Science Foundation (NSF) with a CAREER award in 2009 and by the college of LAS with an Early Achievement in Research Award in 2010, and a Big-12 Fellowship in 2012. He is a senior member of the ACM and of the IEEE.