Remove RATs from Your Code: Automated Optimization of Resource Inefficient Database Writes for Mobile Applications

Yingjun Lyu
University of Southern California, USA

Ding Li
NEC Labs America, USA

William G. J. Halfond
University of Southern California, USA

ABSTRACT

Developers strive to build feature-filled apps that are responsive and consume as few resources as possible. Most of these apps make use of local databases to store and access data locally. Prior work has found that local database services have become one of the major drivers of a mobile device's resource consumption. In this paper we propose an approach to reduce the energy consumption and improve runtime performance of database operations in Android apps by optimizing inefficient database writes. Our approach automatically detects database writes that happen within loops and that will trigger inefficient autocommit behaviors. Our approach then uses additional analyses to identify those that are optimizable and rewrites the code so that it is more efficient. We evaluated our approach on a set of marketplace Android apps and found it could reduce the energy and runtime of events containing the inefficient database writes by 25% to 90% and needed, on average, thirty-six seconds to analyze and transform each app.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Software performance**;

KEYWORDS

Performance optimization; mobile applications; database.

ACM Reference Format:
Yingjun Lyu, Ding Li, and William G. J. Halfond. 2018. Remove RATs from Your Code: Automated Optimization of Resource Inefficient Database Writes for Mobile Applications. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3213846.3213865

1 INTRODUCTION

Mobile devices, such as smartphones and tablets, have become increasingly popular, giving users easy access to millions of apps. As users come to depend more on their devices and apps, the efficiency of these apps in using a device's resources becomes an increasingly important consideration. To earn user loyalty and good reviews, developers strive to build apps that are resource efficient. They want their apps to be responsive, consume less energy, and minimize network usage while still providing innovative services and a high quality user experience. The use of local databases can help developers to provide this functionality. In fact, nearly 60% of all Android apps make use of a local database [51]. However, this widespread usage has also meant that local database services have become one of the major drivers of a mobile device's resource consumption [39].

Although all kinds of database interactions can be resource intensive, certain kinds of operations and usage patterns can be especially problematic. One such pattern, the Repetitive Autocommit Transaction (RAT), has become particularly notorious in the Android developer community [11, 15, 17, 28, 62, 63]. In this pattern, developers repeatedly invoke, in a loop, a statement that modifies the database. Unless the loop has been surrounded by explicit transaction controls, the database will repeatedly create a new transaction for every iteration of the loop. Transactions are very expensive since they require file I/O and database locks. Developer blogs have benchmarked this anti-pattern and found that it can reduce throughput by almost 90%. In our own investigations (see Section 6.4), we found instances of RATs that increased the time and energy expended by a loop by over 900%. RATs are also prevalent, our conservative estimate is that one in five apps that use databases contain this inefficient usage pattern (Section 6.3).

Debugging RATs is challenging for developers and automated analyses. A seemingly obvious strategy for developers is to simply surround loops containing database-modifying statements with explicit transaction controls. However, our investigations found that a database-modifying statement and its enclosing loop are usually not in the same method and developers typically need to inspect call-chains, on average, of at least three methods, but sometime up to twelve, to identify a correct repair (Section 6.4). These long call chains can make it very difficult for developers to manually detect and repair RATs. Automated detection is also complicated because an analysis must reason about all inter-procedural paths leading to database-modifying statements and determine if the statements could be executed inside a loop and outside of transaction controls. Reasoning about transaction controls is especially hard because they can be nested and/or inter-procedural. If a RAT is identified, automated repair is also challenging since it requires precise and context-sensitive identification of references to the database and the insertion of database locking commands that could introduce new deadlocks or unwanted semantics into the mobile app code.

To address this problem, we have designed an automated static analysis based approach for detecting and repairing RATs. Our approach is comprised of three distinct inter-procedural static analyses to: (1) detect the occurrence of the RAT anti-pattern; (2) determine whether the data and control dependencies of a database-modifying statement allow it to be safely optimized; and (3) identify the correct places to insert the necessary transaction code so that,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213865

at runtime, they will not cause new deadlocks. We implemented our approach as a prototype tool and ran it against a set of real world mobile apps. Our approach was effective at improving the performance of the apps: it could reduce the energy and runtime of events containing a RAT by 25% to 90%. The design of our approach also made it fast, as it could carry out the analysis in an average of thirty-six seconds per app. Overall, these results are very positive and indicate that our approach can help developers to detect and repair RATs and improve the efficiency of their apps.

The structure of this paper is as follows. Background information about database operations and transactions are introduced in Section 2. An overview of our approach is given in Section 3. The details of our approach are introduced in Section 4 and Section 5. The results of our evaluation are reported in Section 6 and the threats to the validity of our evaluation are discussed in Section 7. Lastly, we have related work in Section 8 and the conclusion in Section 9.

2 BACKGROUND INFORMATION

SQLite is an embedded local SQL database that reads and writes data directly to local disk files. The Android runtime allows developers to manage an SQLite database using the class `SQLiteDatabase`. Standard APIs are provided in this class to read or change the contents of a database. In the Android app ecosystem, SQLite has emerged as the most common local database service and it is used in over 90% of all Android apps that access a local database service [51]. For this reason, our work in this paper focuses specifically on SQLite.

A *transaction* is a sequence of database operations performed as a single atomic unit of work and is used to ensure data integrity. In SQLite, any command that changes the database has to be done in a transaction. We refer to APIs that can modify a database as *Modification Statement APIs (MSAs)*. Invoking these APIs will automatically start a transaction if one is not already in effect, which is referred to as an *autocommit database change*. We refer to transactions that start automatically in this manner as *implicit* transactions. In SQLite, transactions can also be started and ended manually by calling the APIs `beginTransaction` and `endTransaction`. We refer to transactions that are manually started and ended by these APIs as *explicit* transactions. Explicit transactions can be nested; however, only the outermost transaction will perform commits or rollbacks while the inner ones only keep track of the nesting relationship.

Using transactions consumes many resources as they require expensive operations, such as acquiring and releasing locks, writing and deleting rollback journal files, and making database changes in memory and on disk. Loops are places where the autocommit behavior can cause significant inefficiency. For example, the statement at line 10 in Program 1 will implicitly start ten transactions. With explicit transaction control, these database changes can be executed in a batch more efficiently. However, detecting and repairing this problem is likely to be challenging for developers since, as we mentioned in Section 1, they need to manually examine inter-procedural program paths, identify aliasing, and reason about locks. By design, this problem cannot be solved on the database layer. The reason for this is that the databases are given a limited interface to interact with an app and do not have information about the structure of the code. Therefore, they cannot reason about what would be a safe way to expand transactions or batch updates without running the risk of affecting data integrity. These problems and limitations motivated our decision to use static analyses to analyze the code and use this information to guide us optimizing the autocommit database in an automated manner.

3 OVERVIEW OF THE APPROACH

The goal of our approach is to precisely detect autocommit database changes in loops and safely optimize them. To do this our approach first detects the statements, called RATs, that can make autocommit database changes inside loops. Then our approach identifies the proper insert positions and database variables that will be used to invoke transaction control APIs. Based on the identified results, our approach performs additional analyses to determine which RATs can be safely optimized. For these RATs, the approach inserts transaction control operations and ensures that the database changes are made within the context of a transaction. Our approach can be divided into two phases. In the first phase, our approach uses static analysis to detect the RATs in an Application Under Test (AUT). In the second phase, our approach carries out the optimizations by transforming the RATs so that they properly use the transaction controls. We now explain these two phases in detail.

```
1  public class Example {
2    public void main() {
3      Database db = new Database("a");
4      db.beginTransaction();
5      for(int i=0;i<10;i++)
6        db.insert();
7      db.endTransaction();
8      if(db.isOpen()) {
9        for(int i=0;i<10;i++)
10         db.delete();
11     }
12     else {
13       Helper helper = new Helper();
14       for(int i=0;i<10;i++)
15         helper.updateRecord();
16     }
17   }
18 }
19 public class Helper {
20
21   private Database db_h;
22
23   public Helper() {
24     db_h = new Database("b");
25   }
26
27   public void updateRecord() {
28     db_h.update();
29   }
30 }
```

Program 1: Example code containing RATs.

4 DETECTION

The first phase, detection, is responsible for locating the RATs, as well as the loops containing them. The input of this phase is the AUT and the output is the set of identified RATs and the loops containing them.

4.1 Definition of a Repetitive Autocommit Transaction

We define a *Repetitive Autocommit Transaction (RAT)*, p, as a statement that satisfies the following conditions:

(1) p invokes an MSA.
(2) p is in the body of a loop.
(3) There exists a path in the Control Flow Graph (CFG) from the entry of the CFG to p, in which when p is executed, there is no explicit transaction in effect.

The third condition means that there is no transaction enclosing p, which together with the first condition guarantees that when p is executed, it will start a transaction in autocommit mode. The second condition implies that p will potentially execute multiple times. Although the second condition is not strictly necessary, it indicates that the expensive autocommit mode may be triggered multiple times, since it is inside of a loop, and that there exists the potential for significant savings.

4.2 Detecting RATs

The detection phase analyzes the AUT to identify the location of any RATs it may contain. Detection has two steps. In the first step, our approach identifies the statements meeting the first and second conditions. In the second step, our approach determines if the statements identified meet the third condition.

In the first step, the approach identifies the statements that invoke MSAs inside loops. To do this, our approach first builds a CFG for each method in the AUT and then analyzes the CFGs to identify the control-flow defined regions [56]. By definition, each of these regions corresponds to a loop. Next, the approach scans each loop (region) to find invocations that match the signature of an MSA. Each MSA and its containing loop is the output of the first step. To illustrate this step, consider the example shown in Program 1. There are three loops: lines 5–6, 9–10, and 14–15. Within each loop, there is an MSA: db.insert() at line 6, db.delete() at line 10, and db_h.update() at line 28.

In the second step, the approach analyzes the statements marked in the first step to identify which satisfy the third condition. The output of this step is the set of RATs. The goal of this analysis is to determine if there may be an explicit transaction in effect when an MSA executes. Specifically, the analysis checks if there exists a path from the entry of the method to the MSA where each explicit transaction opened (via a call to `beginTransaction`) has been closed (via a call to `endTransaction`). If such a path exists, then the MSA executed on that path is in autocommit mode (i.e., along that path there is not an explicit transaction that surrounds the MSA).

Thematically, this analysis is similar to detecting resource leakage, e.g., [21, 64]. However, a key difference is that for resource leakage, once the resource-releasing API is called, the corresponding resource is released. This is not true for transaction control in Android SQLite because multiple calls to `beginTransaction` can create nested and accumulated transactions, and only if all the transactions are closed via the exact same number of calls to `endTransaction`, will the MSA executed on that path be in autocommit mode. This challenge motivates the design of the analysis we describe below.

Our approach identifies RATs that satisfy the third condition by performing an iterative analysis over the CFG of each method that contains a marked statement. The analysis is defined as a backwards-may analysis that propagates a set of tuples of the form $\langle$*Statement*, *Counter*$\rangle$ over the edges of the CFG; where *Statement* is one of the statements identified in the first step, and *Counter* is an integer that tracks the number of unmatched invocations of `beginTransaction` and `endTransaction` along the path. The analysis increments *Counter* when it encounters an `endTransaction` and decrements *Counter* when it encounters a `beginTransaction`. Therefore a tuple for which *Counter* is equal to 0 represents a path along which there were an even number of invocations to `beginTransaction` and `endTransaction`. When a tuple arrives at the entry with *Counter* equal to 0, it means that the third condition is true for the statement in the tuple.

The equations for the iterative flow analysis are shown in Figure 1. The Gen set for each of the marked statements (marked in the first step) is initialized to $\langle$statement,0$\rangle$. Then the approach iterates in reverse topological order over the statements in the CFG and updates the In and Out sets. Since the analysis is a backwards-may analysis, the In set of a statement is computed by the union of its successors' Out sets. This iterative process continues until the values of all the Out sets reach a fixed point (i.e., two iterations have the same elements in each node's Out sets). The analysis then terminates. The statements that reach the entry with *Counter* equal to 0 are the RATs.

The role of the Out function is to adjust the counter value and prevent certain tuples from propagating. If s is a `beginTransaction`, then the *Counter* in all of the tuples of the In set will be decreased by one. Note that the condition $i > 0$ is used to prevent tuples with a counter value less than 0 from propagating. The reason for this is that when a tuple whose value is 0 encounters a `beginTransaction`, the analysis can infer two things: (1) all invocations to `beginTransaction` and `endTransaction` along the path have been matched, since the value is 0, and (2) the MSA will be in the transaction explicitly started by the `beginTransaction`. Note that this propagation policy means that any tuple that arrives at the entry with a counter value greater than 0 corresponds to a path with an excess of calls to `endTransaction`. If at runtime, an `endTransaction` is invoked while no transaction is currently open, an exception will be thrown. Assuming that the code does not throw the exception "no transaction is active" at runtime, such tuples will have been caused by a traversal over an infeasible path. If s is an `endTransaction`, then the *Counter* will be increased by one.

The design of our algorithm as described above leaves one open case, namely where an `endTransaction` inside of a loop is used to close multiple explicit transactions started outside of the loop. In the presence of such structure, the counter of the tuple that can reach the `endTransaction` can increase infinitely. To address this situation, we introduce the condition $i < LIMIT$ to ensure that there is an upper bound to which the analysis can converge and reach a fixed point. As with most loop bounded analysis models, this can result in reduced accuracy of our analysis. In this case, the loop bounding can result in false negatives when the limit is set to less than the number of iterations that would actually happen at runtime. This happens because the analysis will identify a `beginTransaction` as being unclosed by an `endTransaction` (since it

under-approximated the loop iterations), and therefore assume that the MSA is part of the transaction opened by the incorrectly unmatched `beginTransaction`. As we discuss in more detail below in Section 4.3, if inaccuracy was necessary, then our design tradeoff was to favor false negatives over false positives.

To perform the analysis inter-procedurally, we use the Cloned Call Graph (CCG) [56] of the AUT instead of the intra-procedural CFG. The CCG contains multiple instances of a method such that every distinct calling context invokes a different instance. Using this context-sensitive cloned call graph, our analysis works the same as if applied to an intra-procedural CFG. The CCG is generally quite large for even small programs. Therefore, we first pruned the call graph of the AUT so that only methods that contained an MSA or a call to `beginTransaction` or `endTransaction`, and their transitive callers remained. Due to the sparsity of database related commands in an AUT, this resulted in the removal of more than 95% on average from our subject applications' CCGs and a dramatic improvement in scalability and efficiency.

We illustrate the analysis using the `main` function of Program 1. The Gen sets for the three marked statements (lines 6, 10, 28) are {⟨db.insert(), 0⟩}, {⟨db.delete(), 0⟩} and {⟨db_h.update(), 0⟩}. The analysis starts at the exit of the CFG at line 17 and is performed backwards. When line 15 is reached, the In set will be empty and the tuple {⟨db_h.update(), 0⟩} will be in its Out set. When line 7 is reached, there will be two tuples, ⟨db.delete(),0⟩ and ⟨db_h.update(),0⟩ in its In set. The *Counter* in the tuples are then increased by one. When line 4, db.beginTransaction(), is reached, the In set contains ⟨db.delete(),1⟩, ⟨db_h.update(),1⟩ and ⟨db.insert(),0⟩. However only the first two will be in the Out set because they have *Counter* value $i > 0$, and they will become ⟨db.delete(),0⟩ and ⟨db_h.update(),0⟩. Finally, because the two tuples can reach the entry of the method `main`, the two statements in the tuples are identified to be satisfying the third condition and are identified as RATs.

$$Gen(s) = \begin{cases} \{\langle s, 0\rangle\}, & \text{if } s \in \text{ (marked statements)} \\ \emptyset, & \text{otherwise} \end{cases}$$

$$In(s) = \bigcup_{p \in succ(s)} Out(p)$$

$$Out(s) = \begin{cases} \{\langle n, i+1\rangle \mid \langle n, i\rangle \in In(s) \wedge i < LIMIT\}, \text{if} \\ s \text{ invokes endTransaction} \\ \{\langle n, i-1\rangle \mid \langle n, i\rangle \in In(s) \wedge i > 0\}, \text{if } s \text{ invokes beginTransaction} \\ Gen(s) \cup In(s), \text{otherwise} \end{cases}$$

Figure 1: Flow analysis

4.3 Analysis Design Accuracy Trade-off

Two typical problems for developers when using a static analysis technique are imprecision (i.e., false positives) and scalability. When we designed our approach, we considered soundness and scalability as our primary goals in order to maximize the practicality of our technique. Therefore, our design favored false negatives over false positives when inaccuracy was unavoidable. In our approach a false-positive is defined as when a statement is identified as a RAT, but it does not actually satisfy the definition. A false positive would cause our approach to introduce an unnecessary transaction, which would potentially increase energy consumption and degrade runtime performance. A false-negative would occur when there exists a statement that meets the definition of a RAT, but it is not identified. Although a false-negative will lead the approach to miss an optimization opportunity, it does not have any other negative effect because the program will remain the same. Since the goal of technique is to optimize where possible and not introduce any change that could consume more energy or reduce performance, we prefer to have false-negatives over false-positives.

Our approach can have false negatives in two situations. The first of these relates to the upper bound on the loop analysis and was discussed in Section 4.2. The second is due to the fact that the approach safely but imprecisely assumes that all database variables alias with each other. This means that if there is a transaction in effect, the flow analysis in Figure 1 considers that the MSA statement is within a transaction no matter which database variable the transaction is initiated by. This can cause false-negatives because the transaction in effect may not belong to the same database to which the MSA statement is writing data. In this case, our analysis misses a RAT. However, our approach is designed this way to maintain scalability when analyzing the CCG. In the current form, our approach does not need to examine the whole program to compute safe and precise aliasing relationships. If an unsafe but precise alias analysis is used, which means two variables can be reported to be not aliased but in fact they are, the alias analysis will lead to a false positive, which violates our design goal. Program 2 shows how an unsafe analysis would introduce false positives. If db1 and db2 are reported to be not aliased, then db1.MSA() will be considered as a RAT because it is not inside any transaction. However, if db1 and db2 are in fact pointing to the same memory location, reporting db1.MSA() as a RAT is a false-positive. Hence, we do not incorporate alias analysis into our flow analysis.

```
1 db2.beginTransaction();
2 for(int i = 0; i < 10; i++)
3    db1.MSA();
4 db2.endTransaction();
```

Program 2: Example code.

We have also identified one special case, shown in Program 3, related to aliasing that would lead to a false positive. In this case, our analysis incorrectly assumes a transaction is ended and the result is a false positive. If db1 and db2 are not aliased, in every possible path that can reach db1.MSA(), the transaction initiated by db1 will be in effect at line 6. However, our analysis will identify it as a RAT because the analysis would consider the transaction opened by db1 to be closed by db2 at line 4, but because of the branch statement at line 1, there exists a path (1F, 3, 4, 5T) in the CFG where db1.MSA() is not inside any unclosed transaction. Note that if db2.beginTransaction() at line 2 is not nested in a conditional statement, db1.MSA() will be considered to be in an unclosed transaction and the detection result will be correct. To understand the potential impact of this false positive, we searched for this

structure in our subject apps. We first designed an analysis to filter out the detected RATs with no invocations to `beginTransaction` and `endTransaction` on the call path and then manually checked the remaining ones. We did not find this interleaving structure in any of the apps. We conclude that while theoretically possible, this represents an unusual coding structure in practice.

```
1 if(db2.isOpen())
2    db2.beginTransaction();
3 db1.beginTransaction();
4 db2.endTransaction();
5 for(int i = 0; i < 10; i++)
6    db1.MSA();
7 db1.endTransaction();
```

Program 3: Example code.

5 OPTIMIZATION

Given the RATs and the loops containing them, the goal of our optimization is to automatically transform the AUT by grouping the RATs' multiple autocommit transactions into a single explicit transaction. At a high-level, our approach must rewrite the loop code so that a transaction is started (i.e., with `beginTransaction`) prior to the loop's execution and then committed (i.e., with `endTransaction`) after the loop finishes execution. There are four major challenges we need to address to carry out this transformation successfully.

The first is to determine the places where the transactional controls can be inserted. The insert positions have to ensure that the transaction control will be carried out successfully without undesirable side-effects. After identifying the proper insert positions, the second challenge is to identify which database reference to use with the transaction controls. A naive approach is to use the database variable in the RAT directly. However, this would not work if a reported RAT and the insert positions are not in the same method. Even if they are, the database variable may not be even initialized at the insert positions. The third challenge is to determine whether inserting transactional controls will cause new deadlocks. Since transactions will lock the database from writing, inserting database locks to the program may interleave the database locks with existing Java locks and cause new deadlock. Finally, the fourth challenge is to guarantee that the inserted transaction can terminate properly, even in the presence of infinite loops and exceptional flow. Otherwise an unterminated transaction not only fails to commit the database changes, but also blocks other transactions from execution.

To address these challenges, we designed an approach to guarantee a successful transformation. The analysis has three phases, each of which corresponds to one or more of the challenges described above. In the first phase, our approach identifies the positions at which the transaction methods would need to be inserted for each RAT. In the second phase, our approach performs an additional analysis to determine whether the database variable accessed in RAT can be safely used and whether optimizing the RAT may cause new deadlocks. In the third phase, our approach transforms the program by inserting the transaction methods into the AUT in a way that ensures the inserted transaction can terminate properly.

5.1 Identifying Insert Positions

The goal of this phase is to identify the positions around the loop where the approach should insert the new transaction-related invocations. The insert positions must ensure that (1) the `beginTransaction` would execute before the loop and the `endTransaction` would execute after the loop finishes; (2) both the `beginTransaction` and `endTransaction` would be guaranteed to execute pairwise; and (3) that all changes made by the RATs in a loop would be grouped into the same transaction. The second condition ensures that all of the changes would be committed and that the transaction method would not cause a runtime exception.

These three conditions can be satisfied by placing the `beginTransaction` at the immediate dominator (begin point) and the `endTransaction` at the immediate post-dominator (end point) of the loop [56]. These points satisfy the above conditions because the begin/end points execute before/after the loop, satisfying condition (1). These two points are unique and any path that can reach the loop has to go through these two points. This guarantees that inserted invocations are pairwise, satisfying condition (2). Moreover, the inserted invocations will open a transaction that remains in effect throughout the loop execution and groups the database changes inside, satisfying condition (3). Our analysis identifies these two points using standard control-flow analysis based techniques [56]. To illustrate, for the RAT at line 10 of Program 1, the begin point would be between lines 8 and 9, and end point would be between lines 10 and 11.

5.2 Identifying Automatically Optimizable RATs

The goal of this phase is to identify which RATs can be automatically and safely optimized. A RAT is an *Automatically Optimizable Repetitive Autocommit Transaction (ARAT)* if (A) the begin and end insertion positions are in the scope of the database variable accessed in the RAT; and (B) the inserted `beginTransaction` and `endTransaction` will not cause new deadlocks. These two conditions are explained in more detail below.

5.2.1 *In-Scope Insert Positions.*

This condition means that a reference to the database object used by the RAT is available at the begin and end points. For example, if a database reference is created inside a loop, then it is not in scope for the potential begin point. This in-scope requirement is necessary because the transaction methods must be called on the same database object that has been identified as part of the hotspot. If a reference is not available at both the begin and end points then the corresponding RAT cannot be optimized.

We now provide the formal definition of in-scope. Given the reference v to the database used in the RAT, the loop L surrounding the RAT, the method m containing L, and the class c where m is defined, the RAT meets the in-scope condition if there exists a variable v' that meets conditions:

(a) One of the following three:
  (1) v' is a local variable declared in m;
  (2) v' is a method parameter of m;
  (3) v' is a class field declared in c or the parent classes of c.
(b) v' and v are must-aliased;

(c) v' is not defined in the loop L, i.e., there does not exist a reaching definition of v' in L;

Broadly, there are three types of references that can serve as v': local variables, class fields, and parameters to the enclosing method. For all three, there are two necessary conditions. First, v and v' must have a must-alias relationship [3]. If two variables are must-aliased, they are aliased regardless of the actual path that is taken at runtime. This means that a transaction invoked using v' will reference the database object pointed to by v. Second, v' cannot be redefined in the loop. The second condition is similar to identifying the loop invariant and is introduced to exclude the situation where at the insert points v' aliases to the database object used by the RAT, but then is assigned to another object inside the loop. Note that these conditions may miss opportunities for optimization but will not incorrectly identify an ARAT. As we explained in Section 4.3, our analysis favors precision over recall.

Standard analyses can be used to identify conditions (a1-a3). For condition (b) and (c), we based the analyses on a reaching definitions analysis [56]. The confluence operator for condition (b), the must-aliased relationship, is set intersection, making it a must analysis and for condition (c), the is-defined relationship, the operator was it union, making it a may analysis.

To illustrate the in-scope analysis, consider the example code in Program 1. The inputs are the statements db.delete() at line 10 and db_h.update() at line 28. For statement db.delete(), we have the variable db in the loop region that starts at line 9. The reaching definition analysis indicates that the only statement that defines db is the statement at line 3, and since line 3 is not inside the same region and db aliases to the local variable db defined at line 3, db.delete() meets the in-scope requirement. Similarly, when db_h is analyzed, the analysis shows the definition statement is at line 13, where the constructor of `Helper` is invoked and db_h is defined in the constructor. Because line 13 is not inside the same region, the region that starts at line 14, db_h is not defined in the region. However, db_h is a field and is declared in class `Helper`. Since the class `Helper` is different from the class of `main`, db_h.update at line 28 does not meet the in-scope requirement and thus is not an ARAT.

5.2.2 Avoiding New Deadlocks. This condition ensures that the insertion of `beginTransaction` and `endTransaction` does not introduce any new possibility of deadlocks. For the transactions themselves, Android SQLite avoids deadlock by making transactions (database locks) mutually exclusive [68]. However the issue of deadlock arises when the AUT contains Java locks (`synchronized` primitives and the `Lock` APIs). Our approach's insertion of a new transaction must not introduce new deadlocks with respect to the existing locks. Prior techniques [8, 14, 52] that focused on lock insertion are not applicable here because they do not handle the situation where there are already locks present in the program. To perform a safe optimization, we based our analysis on the following insight. An insertion that avoids introducing a new deadlock must preserve the consistency of the ordering of the existing lock acquisitions [24]. Here "order" refers to the nesting relationship between locks along a control flow path. For example, lock A nests lock B if lock B is acquired before lock A is released. A safe insertion would therefore not violate any consistent order between database and Java locks already in the AUT.

As part of the process to determine if an insertion can be done while maintaining consistent ordering, our approach checks the following two conditions: (a) a database lock is always the first lock (i.e., there is no Java lock nesting this database lock along any path that can reach the database lock); (b) a database lock is always the last lock (i.e., there is no Java lock nested by this database lock along any path that this database lock can lead to).

Based on these conditions, we describe four scenarios where the RAT represents an ARAT. Note that these scenarios are conservative in that they represent an under-approximation of when it is safe to insert additional locks. The first scenario is when there are no existing database locks in the AUT. The second scenario is when there exists database locks but there are no nested locks (i.e., both (a) and (b) are true). In these two scenarios, a consistent order of locks can be maintained by ensuring the inserted database locks are either always the first or the last. Note that if the inserted lock does not nest and is not nested by any Java lock, the insertion meets this requirement because the inserted database is always the first and the last lock. The third scenario is when there are nested locks and the ordering of these locks is consistent (i.e., (a) xor (b) is true). In this case, a consistent order can be maintained by ensuring that the inserted database lock follows this order. The fourth scenario is when there are nested locks, but the order of these locks is inconsistent (i.e., both (a) and (b) are false). In this case, an insertion of a database lock is safe only if the inserted database lock will not nest or be nested by any Java lock. If none of these scenarios apply, then the approach determines that it cannot safely insert the database transactions.

Our analysis to identify conditions (a) and (b) leverages the same framework as is used in Section 4. For condition (a), to ensure a database lock is always the first lock, our analysis ensures there does not exist any `beginTransaction` that can reach an unreleased Java lock in the backward data flow. For condition (b), to ensure a database lock is always the last lock, our analysis ensures there does not exists any Java lock acquisition statements that can reach an unclosed transaction (i.e., unreleased database lock) in the backward data flow.

```
1  public void run() { //Thread1
2      for(int i=0; i<10; i++) {
3          synchronized (lock){
4              db.insert();
5          }
6      }
7  }
8  public void run() { //Thread2
9      synchronized(lock) {
10         db.beginTransaction();
11         for(int i=0; i<10; i++)
12             db.insert();
13         db.endTransaction();
14     }
15 }
```

Program 4: Example code.

To illustrate the deadlock analysis, consider the example Program 4, the only existing database lock is acquired at line 10 and is nested in the `synchronized` lock, matching the third scenario.

However, the loop starting at line 2 contains a `synchronized` block, which makes inserting a `beginTransaction` introduce an inconsistent order. Therefore, db.insert() at line 4 is not optimizable. In our example Program 1, the only existing database lock at line 4 does not nest and is not nested in any Java locks, matching the second scenario. The in-scope RAT, db.delete() at line 10, is an ARAT because inserting a database lock at line 9 does not create nested locks.

5.3 Optimizing the Application Under Test

This phase carries out the transformation of the AUT. To do this, our optimization uses each of the database variables v' identified in the second phase (Section 5.2) to invoke transaction-related APIs. For each v', our approach (1) creates an invocation statement v'`.beginTransaction()` and inserts it after the begin point, and (2) creates two statements, v'`.setTransactionSuccessful()` and v'`.endTransaction()` and inserts them before the end point. (The API `setTransactionSuccessful` is used to mark the current transaction as successful so that the changes can be committed to the database.)

Exception Handling: To maintain the original semantics of the program when exceptional behaviors are present, our approach wraps the target loop into a `try` block and places the inserted v'`.setTransactionSuccessful()` and v'`.endTransaction()` into a `finally` block. Therefore, if there is an exception thrown during the execution of the target loop, the transaction will still be properly ended and the changes made before the exception will be committed to the database, which matches the unoptimized program's semantics. If the loop is originally in a `try` block, our approach does not add an additional `try` but add the calls to `setTransactionSuccessful` and `endTransaction` to the corresponding `catch` blocks. To do this, our approach creates a local database variable db_copy at the method entry, inserts db_copy=v' after v'`.beginTransaction()`, and db_copy=`null` after v'`.endTransaction()`. In the `catch` block, the added invocations will only be invoked if db_copy!=`null`.

Avoid Lengthy Loops Blocking Concurrent Writes: Our optimization can affect the execution of concurrent database writes. The reason is that after the optimization, the loop will take the lock of the database and hold it until it ends. If there are other threads waiting to write to the database, they will queue up and have to wait. Starvation might occur if the optimized loop holds the lock for a long period of time. To alleviate this problem, our approach inserts a timer in the loop that ends the transaction and opens a new one when the timer has exceeded a predetermined value. By doing so, the blocked transactions can have opportunities to execute. Although this mechanism may introduce an additional occasional transaction cost, this cost is still a net reduction as compared to the original unoptimized app since in the unoptimized version the implicit transactions are triggered in every loop iteration and the cost of retrieving time stamps is trivial compared to the cost of database operations (1/4,300 of the cost of an implicit transaction.)

Other than the starvation problem, some parts of the program may be slowed by the queueing time, but overall, as we show in our evaluation, the program will run faster due to the reduced write time. It is also worth noting that the SQLite database is not designed for highly concurrent database changes [67], so this makes it unlikely that most apps would experience this kind of behavior.

6 EVALUATION

We carried out an evaluation of our approach to measure its precision, impact, and analysis time. We considered the following research questions:

- RQ 1: What is the precision of our approach in identifying ARATs?
- RQ 2: How often do ARATs occur in marketplace apps?
- RQ 3: How much runtime can be reduced by using our approach?
- RQ 4: How much energy can be saved by using our approach?
- RQ 5: What is the analysis time of our approach?

6.1 Implementation

We implemented a prototype of our approach, RAT-TRAP, and used it to carry out our evaluation. Our approach would ideally be integrated into a developer's tool chain and assist them in guiding source level transformation. However, our implementation analyzes and rewrites bytecode to enable the use of a broader and more representative set of subject apps. The prototype is 13K lines of code and leverages Dexpler [6] to convert Dalvik bytecode to Jimple, and Soot [36] to build data structures, such as the CFG and Call Graph (CG). RAT-TRAP also uses the Soot framework to rewrite apps and convert the Jimple to Dalvik bytecode. Note that the front-end parser of Soot could be changed to analyze source code.

6.2 Subject Apps

To obtain subjects, we used the Google Play Crawler [2] to download a random set of 1,887 apps from the Google Play marketplace. The subject apps varied in code size: 15% of them had less than 10K bytecodes, 54% of them had bytecode counts between 10K and 100K, and 31% of them had more than 100K bytecodes. The subject apps were from 23 of the 24 app category types defined by the Google Play marketplace. For all experiments, we ran the subject apps on a Samsung Galaxy S5 smartphone running Android 5.0.

6.3 Experiment 1: Precision and Pervasiveness

The goal of this experiment was to address RQ 1 and RQ 2. To carry out this experiment, we first ran RAT-TRAP on all of the subject apps. For each ARAT detected, we manually inspected the code of the methods in the call chain originating from the MSA to determine if the report was a true positive. From the results of this analysis, we then calculated the precision of the detection analysis and the frequency of occurrence of ARATs in the subject apps. We did not calculate recall as it was not possible to accurately manually analyze all of the bytecode and calculate several types of information, including inter-procedural aliasing information for all 1,887 subject apps.

Our manual analysis verified that each of the reports was a true positive, for a precision of 100%. In terms of prevalence, our analysis found ARATs in 10.9% of the subject apps (206 of 1,887). This rate doubled to about one in five, if we considered only those apps that actually used a database (only 54% of the subject apps accessed a SQLite database). For apps that had an ARAT, the average number of ARATs was five, with a median of two. However, we did find one app that contained 4,220 ARATs! (We did not include this app in our

reported statistics.) Overall, these results show that ARATs occur in a high proportion of marketplace apps and that our approach can detect them with high precision.

Our analysis also allowed us to collect several statistics about the structure of ARATs. We found that 44% of the ARATs were interprocedural (i.e., loop headers and MSAs were in different methods) and the average number of methods in the call chains from an ARAT to the call graph entry was 3.3, with the maximum number being 12. We also found that in 11% of the detected ARATs, the corresponding loops nested or were nested by a Java lock. Taken together, these statistics underscore the challenges developers would face in manually identifying and fixing ARATs.

6.4 Experiment 2: Energy Savings and Runtime Improvement

The goal of this experiment is to address RQ 3 and RQ 4. To do this, we measured and compared an app's resource usage, in terms of its runtime and energy consumption, before and after it was optimized by RAT-TRAP. Our experiment used a subset of the subject apps because our protocol required us to individually analyze, interact with, and measure each of the subjects in the experiment.

6.4.1 Protocol. A key challenge in the experiment protocol was to create reasonable workloads for the subjects. Typical techniques, such as random crawling (e.g., using PUMA [23] or Monkey [1]), were unable to generate the specific workload needed to trigger each app's ARAT and the values supplied as inputs that could influence the ARAT might not be representative of typical usage. To address this issue, we selected subject apps for which it would be possible to control the number of iterations of the loop that contained the ARAT. This allowed us to supply a range of input values that was likely to include typical values.

To identify subject apps that satisfied this constraint, we began by choosing an app at random from our subject pool. Then we manually analyzed the call chain containing the ARAT to determine if the ARAT could be triggered by user interaction (e.g., the chain's root method was an event handler for a UI element) and if the input controlling the loop's iterations originated from the user. The manual analysis was verified by running and interacting with an instrumented version of the app. The first six apps to satisfy this constraint were used in our experiment. Detailed information about these apps is shown in Table 1. For each app, this table provides a description of its key functionality and the number of bytecodes in the CCG (# Rel-bytecode).

For each of the apps, we created a set of workloads that led to the execution of the optimized ARAT. Each workload consisted of two phases, setup and triggering. For the setup phase, we generated a sequence of actions that provided inputs to the app. For example, the input for App180 was the number of quizzes desired. (We explain how we chose these values in more detail below.) In the triggering phase, we performed the specific actions that triggered the identified event. For example, for App180, clicking a button to retrieve the quizzes caused them to be written to the database, which was the code that contained the ARAT. The column labeled "Event" in Table 1 describes the triggering event we identified for each of the subject apps.

To determine the input values for the setup phase, we interacted with the app and determined a reasonable minimum and maximum for the inputs based on the app's functionality and the layout of the app's user interface. Within that range, we then chose four roughly equidistant points and built a workload that used each of those values as inputs in the setup phase. In Table 1 we list the input numbers under the column "Inputs". We recorded each workload using the Reran [18] tool. We then replayed this workload on the optimized and unoptimized version of the instrumented app. The use of Reran minimized the potential for variations in the execution of the workload between versions. For example, Reran ensured that inputs for both versions were entered in the exact same amount of time.

During the execution of the workloads, we measured the resource usage of the optimized and unoptimized versions of the apps at two levels of granularity. The first measurement was at the level of the individual loop that had been the target of the optimization. We targeted this level of granularity because it provided the most localized context for the resource usage. The second was at the app level/callback level (e.g., onClick). We targeted this level because it quantified the resource changes in the context of an event that would be visible to an end user.

To record the resource usage at these two levels of granularity, we instrumented the optimized and unoptimized apps with timestamps. For the loop-level measurement, we inserted timestamps before and after each target loop. For the app level measurements, we inserted timestamps at the beginning of the root event method as well as the spots where the method exited. To determine the runtime for each version of the app at each level of granularity, we calculated the difference between the starting and ending timestamps. To calculate the energy consumption we ran the apps while the smartphone was connecting to a Monsoon power meter [55] with a clock synchronized to that of the smartphone and then summed the energy consumed between the starting and ending times. For each input, we repeated the measurements fifteen times to reduce the impact of non deterministic or uncontrolled behavior. Therefore, for the optimized and unoptimized version of an app, we took a combined 120 measurements. Between each measurement, we added a waiting time to avoid any impact from tail energy behavior and to allow the device to cool down. We ran a Mann-Whitney U test ($\alpha = .05$) to compare the differences in measurements between the unoptimized version and optimized version for both the energy and runtime measurements. The results of this test established statistical significance for all of the results of our subjects.

6.4.2 Result. The results of our experiment are shown in Table 2 and Figure 2. For each subject app, Table 2 shows the median values of the runtime and energy consumption of the loops and events that contained the ARATs. In each of the last four columns, the value of the unoptimized version is shown on the left and the value of the optimized version is shown on the right. Figure 2 shows the runtime and energy consumption reduction data in Table 2 as percentages. In each sub-figure, the Y axis shows the percent reduction of the optimized app versus the unoptimized app. The X axis shows the indexes of input numbers corresponding to the "Input Number" column in Table 1. Note that the runtime and energy consumption

Table 1: Description of subject apps and workload information

App	Name	Description	# Rel-bytecode	Event	Inputs
App69	PayAnyWhere	Mobile Payment Processing	20,134	Record Items Sold	5, 10, 15, 20
App180	ASVAB Test	ASVAB Practice Quiz	744	Select Quiz Questions	10, 20, 30, 40
App219	Nail Games	Nail Decoration	415	Save a Nail with Decorations	3, 5, 10, 15
App567	CATA	Bus Schedule Checker	2,197	Update Favorite Lists	3, 5, 10, 15
App802	Tech News	Technology Newspaper	919	Add Newspapers to Update List	3, 5, 10, 15
App1447	Ride Chicago	Transportation Schedule Checker	4,553	Add Trains to Favorite	3, 5, 8, 10

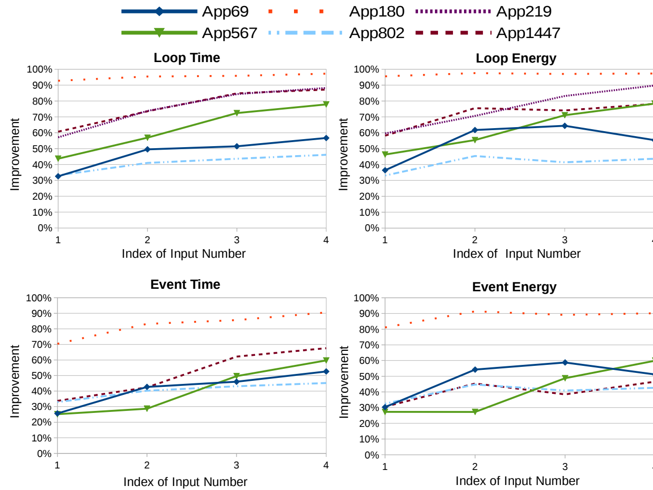

Figure 2: Percent reduction in energy and runtime.

Table 2: Median runtime (ms) and energy consumption (mJ) of the unoptimized/optimized version

App	Input	Loop Time	Loop Energy	Event Time	Event Energy
App69	5	69/45	36/20	88/62	48/30
	10	125/63	86/35	144/84	101/49
	15	182/90	183/65	201/109	197/82
	20	238/104	235/107	258/121	264/126
App180	10	172/12	505/20	249/69	672/119
	20	410/17	1472/29	481/76	1747/142
	30	515/21	2092/61	583/84	2356/257
	40	872/23	3017/78	951/86	3121/303
App219	3	9/4	4/1	695/694	769/753
	5	14/3	6/1	700/691	777/758
	10	27/4	12/1	719/692	790/771
	15	40/5	20/2	727/695	817/764
App567	3	17/9	14/7	47/29	41/25
	5	25/10	20/8	55/38	44/32
	10	46/11	42/10	73/33	68/31
	15	67/14	56/12	93/36	80/30
App802	3	31/22	15/12	32/22	16/12
	5	50/31	25/15	51/31	26/15
	10	73/44	42/24	74/45	42/25
	15	100/54	54/28	101/54	54/28
App1447	3	26/10	14/5	74/46	38/26
	5	40/11	21/6	78/51	46/27
	8	72/12	42/10	123/50	77/47
	10	95/12	48/11	147/47	75/42

improvement for App219 are not shown in the app level graphs. This is because although the runtime and energy consumption was reduced at the loop level, they did not differ by a wide enough margin at the app level for it to be visible in the graph.

6.4.3 Discussion of Results. The results show that our approach was able to achieve significant runtime improvements at both the loop and app level. As shown in Figure 2, the average loop execution time was reduced 66% and ranged from 30% to 97% across the different subjects. The average event execution time was reduced 51% and ranged from 25% to 90%. From the figures, it can be seen that the amount of runtime improvement was generally proportional to the size of the input (i.e., the number of loop iterations). The reason for this is that the larger the input number, the more transactions were initiated in the unoptimized version of an app. But after the optimization only one transaction was initiated per loop regardless of the size of the input. Although the savings were higher for higher number of loop iterations, it is important to note that even at the app level, savings of over 25% were typical when the number of loop iterations was low. This shows that the savings realized by the optimization can be significant even for apps where the number of iterations of a loop with ARATs are expected to be small.

The runtime improvements at the app level are particularly important because they show that unnecessary database transactions can severely undermine the responsiveness of an app. Research on humans' perception of app responsiveness has suggested that 100 ms is the threshold for humans to perceive delay [53, 57]. As can be seen in Table 2, in four of the six subject apps, the runtime for a targeted event at an input number went from over 100 ms in the unoptimized version to under 100 ms. These results indicate that our approach can help reduce human perceivable delays.

The energy savings achieved by our optimization were also significant. As shown in Figure 2, the loop level energy consumption was reduced 36% to 97% and averaged 67%. The app level energy consumption was reduced 27% to 90% and averaged 51%. As with the runtime, the larger the input number, the more energy was saved by the optimization. Although the percentages of savings at the app level were lower than the ones at the loop level, they were still very high, which showed that significant local reductions in energy consumption can result in app level savings. The reduction in energy consumption also showed that our technique did not speed up the code at the cost of consuming more resources, e.g., CPU or memory. Instead, our technique can improve runtime as well as save energy by reducing the consumption of resources.

6.5 Experiment 3: Analysis Time

To answer RQ 5, we evaluated how much time was needed to run RAT-TRAP on a typical app. We measured the time spent analyzing and rewriting the six apps we used in Experiment 2. For each app, we ran our tool fifteen times and calculated its average execution time. The total execution time ranged from 18 seconds to 96 seconds with an average of 36 seconds and median of 25 seconds. Generally, most of the total execution time (87%), was consumed by converting bytecodes from one format to another. The detection analysis time was around or under 2 seconds, except for App69, which was 51 seconds. This was due to this app having a large amount of code relevant to the database and our detection analysis needed to analyze a relatively large CCG. Among the broader pool of subject apps, less than 2% had this amount of relevant code. The rewrite time was less than 0.1 seconds. Taken together these numbers suggest that the runtime of our approach would not be a barrier to its acceptance by developers.

7 THREATS TO VALIDITY

External Validity: Using only six subjects may introduce threats to the external validity of our results, if the selected apps are not representative. To make the selected apps representative, we selected a subset of apps that have at least 10,000 downloads from the Google Play store. Furthermore we also selected apps with different sizes, from 0.5 MB to 30 MB. The apps covered general types of usages of the local database, e.g., insert, update, delete and read.

Internal Validity: In our study, we used *Dexpler* to perform the program transformation. The conversion between Java bytecode and Dalvik bytecode may introduce differences between the transformed version and the original version of the app that could influence runtime performance and energy consumption. To make the effects consistent, we ran *Dexpler* on both the unoptimized and optimized versions.

Construct Validity: A potential threat to the validity of our results is that our measured savings might not directly reflect an end user' savings for their particular usage of the app. Although realistic user traces could partly address this threat, we did not have access to real users of the apps, so our trace would still result in a threat to validity. Therefore, we decided to measure at the user event level, which, essentially, represents the smallest unit of interaction with an app that a user could trigger and that would lead to the ARAT. For example, a button press. As a result, a real user's savings would depend on how often they would trigger that event in the course of their typical usage.

8 RELATED WORK

Optimizing the performance of database operations is a classic and well studied problem. Many approaches have been proposed to improve performance by optimizing queries to the database [29, 34, 65]. Researchers have also focused on analyzing and optimizing storage mechanisms for database operations [30–33, 60, 61]. For example, Jeong and colleagues [31] improved the performance of I/O operations by coordinating the different database journaling modes with different filesystems. Oh and colleagues [60] leveraged the phase change memory technique to reduce redundant pages writes. Although this area of research has made important advancements, it neglects the role that software engineering practices and program structure can also have on the resource consumption of database services on mobile devices.

Many researchers have focused on performance optimization for mobile apps. One group of optimization techniques characterized and detected performance and energy bugs, e.g., resource leakage issues in mobile apps [4, 5, 16, 21, 47–50, 64, 70]. A recent study [51] carried out an empirical study of database operations that included energy measurements and leveraged a string analysis technique [42] to analyze database queries. The study also included an analysis of implicit autocommit transactions [51]. However, it did not include algorithms for detection or repair. Besides performance optimization, researchers have also proposed techniques for estimating performance for smartphone applications to aid debugging and optimization [35, 59]. However, none of these approaches targeted optimizing runtime performance of database operations.

Much research effort has also been devoted to energy optimization for mobile apps. One group of techniques optimized energy consumption of mobile apps by bundling HTTP requests [37, 38, 41]. Another group of techniques optimized display energy consumption of mobile apps by using energy efficient color schemes [13, 43, 46, 69]. Nikzad and colleagues [58] proposed a technique to schedule tasks on mobile apps in a more energy efficient way. Zhu and colleagues [71] proposed a set of language extensions to guide energy optimizations in mobile Web applications. Cito and colleagues [10] optimized recurrent advertisement and analytics requests to save energy. However, these technique did not optimize energy consumption of database operations.

There is also a large body of work focusing on energy measurement [7, 12, 22, 25, 27, 40, 54, 64] and many researchers have conducted empirical studies to find potential areas to optimize energy consumption of mobile apps [9, 19, 20, 26, 39, 44, 45, 51, 66]. This paper was inspired by the results of these studies. However, the measurement approaches and the empirical studies cited here did not propose techniques to optimize mobile apps.

9 CONCLUSION

Energy consumption and UI responsiveness are critical for mobile apps. Current approaches for mobile performance or energy optimization do not target inefficient database operations. In this paper we propose an approach to optimize inefficient database writes. Our approach automatically detects the database writes that happen within loops and that will trigger inefficient autocommit behaviors. Our approach then uses additional analyses to identify those that are optimizable and rewrite the code so that it is more efficient. We evaluated our approach on a set of marketplace Android apps. In this evaluation, our approach finished the detection and optimization in less than forty seconds on average and it could reduce the energy and runtime of events containing a RAT by 25% to 90%. Our artifact is available under the Apache Software License v2.0 from https://github.com/USC-SQL/RAT-TRAP.

10 ACKNOWLEDGMENTS

This work was supported, in part, by the National Science Foundation under grant number CCF-1321141.

REFERENCES

- [1] [n. d.]. Android Monkey. http://goo.gl/wSlG0b.
- [2] Akdeniz. 2017. Google Play Crawler. http://goo.gl/0yDL5w.
- [3] Rita Z. Altucher and William Landi. 1995. An Extended Form of Must Alias Analysis for Dynamic Allocation. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 74–84. https://doi.org/10.1145/199448.199466
- [4] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting Energy Bugs and Hotspots in Mobile Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 588–598. https://doi.org/10.1145/2635868.2635871
- [5] Abhijeet Banerjee and Abhik Roychoudhury. 2016. Automated Re-factoring of Android Apps to Enhance Energy-efficiency. In *Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft '16)*. ACM, New York, NY, USA, 139–150. https://doi.org/10.1145/2897073.2897086
- [6] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis (SOAP '12)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2259051.2259056
- [7] R.J. Behrouz, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann. 2015. EcoDroid: An Approach for Energy-Based Ranking of Android Apps. In *Green and Sustainable Software (GREENS), 2015 IEEE/ACM 4th International Workshop on*. 8–14. https://doi.org/10.1109/GREENS.2015.9
- [8] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. 2008. Inferring Locks for Atomic Sections. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 304–315. https://doi.org/10.1145/1375581.1375619
- [9] Shaiful Alam Chowdhury, Varun Sapra, and Abram Hindle. 2015. Is HTTP/2 more energy efficient than HTTP/1.1 for mobile users? *PeerJ PrePrints* 3 (2015), e1571.
- [10] Jürgen Cito, Julia Rubin, Phillip Stanley-Marbell, and Martin Rinard. 2016. Battery-aware Transformations in Mobile Applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 702–707. https://doi.org/10.1145/2970276.2970324
- [11] Mike Dalisay. 2015. Android SQLite Transaction Example with INSERT Prepared Statement. https://www.androidcode.ninja/android-sqlite-transaction-tutorial/.
- [12] Mian Dong, Yung-Seok Kevin Choi, and Lin Zhong. 2009. Power Modeling of Graphical User Interfaces on OLED Displays. In *DAC*.
- [13] Mian Dong and Lin Zhong. 2012. Chameleon: A Color-Adaptive Web Browser for Mobile OLED Displays. *IEEE Transactions on Mobile Computing* (2012).
- [14] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. 2007. Lock Allocation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 291–296. https://doi.org/10.1145/1190216.1190260
- [15] Jason Feinstein. 2017. Squeezing Performance from SQLite: Insertions. https://medium.com/@JasonWyatt/squeezing-performance-from-sqlite-insertions-971aff98eef2.
- [16] A. Ferrari, D. Gallucci, D. Puccinelli, and S. Giordano. 2015. Detecting energy leaks in Android app with POEM. In *Pervasive Computing and Communication Workshops (PerCom Workshops), 2015 IEEE International Conference on*. 421–426. https://doi.org/10.1109/PERCOMW.2015.7134075
- [17] William J. Francis. 2013. Turbocharge your SQLite inserts on Android. https://www.techrepublic.com/blog/software-engineer/turbocharge-your-sqlite-inserts-on-android/.
- [18] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. 2013. RERAN: Timing- and touch-sensitive record and replay for Android. In *Software Engineering (ICSE), 2013 35th International Conference on*. 72–81. https://doi.org/10.1109/ICSE.2013.6606553
- [19] Jiaping Gui, Ding Li, Mian Wan, and William G.J. Halfond. 2016. Lightweight Measurement and Estimation of Mobile Ad Energy Consumption. In *Proceedings of the International Workshop on Green and Sustainable Software (GREENS)*.
- [20] Jiaping Gui, Stu Mcilroy, Mei Nagappan, and William G. J. Halfond. 2015. Truth in Advertising: The Hidden Cost of Mobile Ads for Software Developers. In *ICSE*.
- [21] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. 2013. Characterizing and detecting resource leaks in Android applications. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. 389–398. https://doi.org/10.1109/ASE.2013.6693097
- [22] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. 2013. Estimating Mobile Application Energy Consumption Using Program Analysis. In *ICSE*.
- [23] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*. ACM, New York, NY, USA, 204–217. https://doi.org/10.1145/2594368.2594390
- [24] James W. Havender. 1968. Avoiding Deadlock in Multitasking Systems. *IBM Systems Journal* 7, 2 (1968), 74–84.
- [25] Abram Hindle. 2012. Green mining: A methodology of relating software change to power consumption. In *MSR*. IEEE Press, 78–87.
- [26] Abram Hindle. 2015. Green Mining: A Methodology of Relating Software Change and Configuration to Power Consumption. *Empirical Softw. Engg.* 20, 2 (April 2015), 374–409. https://doi.org/10.1007/s10664-013-9276-6
- [27] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. 2014. GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 12–21. https://doi.org/10.1145/2597073.2597097
- [28] Espen Hovlandsdal. 2011. Speeding up SQLite insert operations. http://bytes.schibsted.com/speeding-up-sqlite-insert-operations/.
- [29] Matthias Jarke and Jurgen Koch. 1984. Query Optimization in Database Systems. *ACM Comput. Surv.* 16, 2 (June 1984), 111–152. https://doi.org/10.1145/356924.356928
- [30] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. 2013. AndroStep: Android Storage Performance Analysis Tool.. In *Software Engineering (Workshops)*, Vol. 13. 327–340.
- [31] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX, San Jose, CA, 309–320. https://www.usenix.org/conference/atc13/technical-sessions/presentation/jeong
- [32] H. Kim, M. Lee, W. Han, K. Lee, and I. Shin. 2011. Aciom: Application characteristics-aware disk and network I/O management on Android platform. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*. 49–58.
- [33] Je-Min Kim and Jin-Soo Kim. 2012. *Frontiers in Computer Education*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter AndroBench: Benchmarking the Storage Performance of Android-Based Mobile Devices, 667–674. https://doi.org/10.1007/978-3-642-27552-4_89
- [34] Jonathan J. King. 1981. QUIST: A System for Semantic Query Optimization in Relational Databases. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7 (VLDB '81)*. VLDB Endowment, 510–517. http://dl.acm.org/citation.cfm?id=1286831.1286881
- [35] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. 2013. Mantis: Automatic Performance Prediction for Smartphone Applications. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 297–308. http://dl.acm.org/citation.cfm?id=2535461.2535498
- [36] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*.
- [37] Ding Li and William G. J. Halfond. 2014. An Investigation Into Energy-Saving Programming Practices for Android Smartphone App Development. In *GREENS*.
- [38] Ding Li and William G. J. Halfond. 2015. Optimizing Energy of HTTP Requests in Android Applications. In *Proceedings of the Third International Workshop on Software Development Lifecycle for Mobile (DeMobile) – Short Paper*.
- [39] Ding Li, Shuai Hao, Jiaping Gui, and William Halfond. 2014. An Empirical Study of the Energy Consumption of Android Applications. In *ICSME*. IEEE.
- [40] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. 2013. Calculating Source Line Level Energy Information for Android Applications. In *ISSTA*.
- [41] Ding Li, Yingjun Lyu, Jiaping Gui, and William G.J. Halfond. 2016. Automated Energy Optimization of HTTP Requests for Mobile Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*.
- [42] Ding Li, Yingjun Lyu, Mian Wan, and William G. J. Halfond. 2015. String Analysis for Java and Android Applications. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
- [43] Ding Li, Huyen Tran, Angelica, and G. J. Halfond, William. 2014. Making Web Applications More Energy Efficient for OLED Smartphones. In *ICSE*.
- [44] Jing Li, Anirudh Badam, Ranveer Chandra, Steven Swanson, Bruce L Worthington, and Qi Zhang. 2014. On the energy overhead of mobile storage systems.. In *FAST*. 105–118.
- [45] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining energy-greedy API usage patterns in Android apps: an empirical study. In *MSR*.
- [46] Mario Linares-Vásquez, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2015. Optimizing Energy Consumption of GUIs in Android Apps: A Multi-objective Approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/2786805.2786847
- [47] Yepang Liu, Chang Xu, and S.C. Cheung. 2013. Where has my battery gone? Finding sensor related energy black holes in smartphone applications. In *PerCom*.

[48] Yepang Liu, Chang Xu, S.C. Cheung, and Jian Lu. 2014. GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. *Software Engineering, IEEE Transactions on* 40, 9 (Sept 2014), 911–940. https://doi.org/10.1109/TSE.2014.2323982

[49] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1013–1024. https://doi.org/10.1145/2568225.2568229

[50] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. 2016. Understanding and Detecting Wake Lock Misuses for Android Applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 396–409. https://doi.org/10.1145/2950290.2950297

[51] Yingjun Lyu, Jiaping Gui, Mian Wan, and William G.J. Halfond. 2017. An Empirical Study of Local Database Usage in Android Applications. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. To Appear.

[52] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. 2006. Autolocker: Synchronization Inference for Atomic Sections. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 346–358. https://doi.org/10.1145/1111037.1111068

[53] Robert B. Miller. 1968. Response Time in Man-computer Conversational Transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I (AFIPS '68 (Fall, part I))*. ACM, New York, NY, USA, 267–277. https://doi.org/10.1145/1476589.1476628

[54] Radhika Mittal, Aman Kansal, and Ranveer Chandra. 2012. Empowering Developers to Estimate App Energy Consumption. In *ACM Mobicom*. ACM. http://research.microsoft.com/apps/pubs/default.aspx?id=166288

[55] Inc Monsoon Solutions. 2017. Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor.

[56] Steven S. Muchnick. 1997. *Advanced compiler design implementation*. Morgan Kaufmann.

[57] Brad A. Myers. 1985. The Importance of Percent-done Progress Indicators for Computer-human Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '85)*. ACM, New York, NY, USA, 11–17. https://doi.org/10.1145/317456.317459

[58] Nima Nikzad, Octav Chipara, and William G. Griswold. 2014. APE: An Annotation Language and Middleware for Energy-efficient Mobile Application Development. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 515–526. https://doi.org/10.1145/2568225.2568288

[59] Adrian Nistor and Lenin Ravindranath. 2014. SunCat: Helping Developers Understand and Predict Performance Problems in Smartphone Applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*.

[60] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. 2015. SQLite Optimization with Phase Change Memory for Mobile Applications. In *Proc. VLDB Endow.*

[61] Hamza Ouarnoughi, Jalil Boukhobza, Pierre Olivier, Loic Plassart, and Ladjel Bellatreche. 2013. Performance analysis and modeling of SQLite embedded databases on flash file systems. *Design Automation for Embedded Systems* 17, 3 (2013), 507–542. https://doi.org/10.1007/s10617-014-9149-2

[62] Stack Overflow. 2013. Android SQLite database: slow insertion. https://stackoverflow.com/questions/3501516/android-sqlite-database-slow-insertion.

[63] Stack Overflow. 2017. Android Sqlite Performance. https://stackoverflow.com/questions/28188164/android-sqlite-performance.

[64] Abhinav Pathak, Abhilash Jindal, Y Charlie Hu, and Samuel P Midkiff. 2012. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys*.

[65] Giuseppe Procaccianti, Héctor Fernández, and Patricia Lago. 2016. Empirical Evaluation of Two Best Practices for Energy-efficient Software Development. *J. Syst. Softw.* 117, C (July 2016), 185–198. https://doi.org/10.1016/j.jss.2016.02.035

[66] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause. 2014. How Does Code Obfuscation Impact Energy Usage?. In *ICSME*.

[67] SQLite. 2017. Appropriate Uses For SQLite. http://www.sqlite.org/whentouse.html.

[68] SQLite. 2017. File Locking And Concurrency In SQLite. http://www.sqlite.org/lockingv3.html.

[69] Mian Wan, Yuchen Jin, Ding Li, and William G. J. Halfond. 2015. Detecting Display Energy Hotspots in Android Apps. In *ICST*.

[70] Yixue Zhao, Marcelo Schmitt Laser, Yingjun Lyu, and Nenad Medvidovic. 2018. Leveraging Program Analysis to Reduce User-Perceived Latency in Mobile Applications. *International Conference on Software Engineering* (2018).

[71] Yuhao Zhu and Vijay Janapa Reddi. 2016. GreenWeb: Language Extensions for Energy-efficient Mobile Web Computing. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 145–160. https://doi.org/10.1145/2908080.2908082