Detecting Cache-Related Bugs in Spark Applications

Hui Li, Dong Wang∗
State Key Lab of Computer Sciences, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
{lilihui2012,wangdong18}@otcaix.iscas.ac.cn

Yu Gao
Wensheng Dou†
Lijie Xu
State Key Lab of Computer Sciences, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
{gaoyu15,wsdou,xulijie}@otcaix.iscas.ac.cn

Tianze Huang∗
Beijing University of Posts and Telecommunications
Beijing, China
huangtianze@bupt.edu.cn

Wei Wang
Jun Wei
Hua Zhong
State Key Lab of Computer Sciences, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
{wangwei,wj,zhonghua}@otcaix.iscas.ac.cn

ABSTRACT
Apache Spark has been widely used to build big data applications. Spark utilizes the abstraction of Resilient Distributed Dataset (RDD) to store and retrieve large-scale data. To reduce duplicate computation of an RDD, Spark can cache the RDD in memory and then reuse it later, thus improving performance. Spark relies on application developers to enforce caching decisions by using persist() and unpersist() APIs, e.g., which RDD is persisted and when the RDD is persisted / unpersisted. Incorrect RDD caching decisions can cause duplicate computations, or waste precious memory resource, thus introducing serious performance degradation in Spark applications. In this paper, we propose CacheCheck, to automatically detect cache-related bugs in Spark applications. We summarize six cache-related bug patterns in Spark applications, and then dynamically detect cache-related bugs by analyzing the execution traces of Spark applications. We evaluate CacheCheck on six real-world Spark applications. The experimental result shows that CacheCheck detects 72 previously unknown cache-related bugs, and 28 of them have been fixed by developers.

KEYWORDS
Spark, cache, bug detection, performance

1 INTRODUCTION
Apache Spark [50, 51] has become one of the most popular big data processing systems, and has been widely used in many big Internet companies, e.g., Facebook [18], Alibaba [2], eBay [19]. As an in-memory big data framework, Apache Spark can provide order of magnitude of performance speedup over disk-based big data frameworks, e.g., Apache Hadoop [1]. Especially, Apache Spark works better on iterative analytics, e.g., machine learning [37], and graph computation [33], which requires data to be shared across different iterations.

A Spark application consists of two kinds of operations: transformation (e.g., map and filter) and action (e.g., count and take). Spark utilizes the abstract of Resilient Distributed Dataset (RDD) to store and retrieve large-scale data, which is a read-only collection of objects partitioned across a set of machines. Transformations are used to create RDDs, and actions are used to obtain actual computation results on RDDs. Once an action completes, all its used RDDs are discarded from memory. To reuse an RDD loaded by an action, Spark provides persist() API to manually cache the loaded RDD. Thus, the following actions can reuse the cached RDD, and avoid duplicate computations. Note that, an RDD can be persisted in different cache levels, e.g., in memory and on disk. Spark provides unpersist() API to manually release cached RDDs when they

1We use cache and persist interchangeably in this paper.
will not be used by following actions, thus saving precious memory. Figure 1a shows an example about how to reuse RDD words (Section 2.1).

Apache Spark relies on developers to enforce caching decisions through `persist()` and `unpersist()` APIs. Developers need to clearly figure out which RDD should be cached, when the RDD is persisted and unpersisted. The lazy evaluation of RDDs in Spark (Section 2.3) further complicates cache mechanisms, e.g., a `persist()` operation must be performed before its corresponding `action` operation. Improper caching decisions on RDDs usually cause performance degradation, even worse, out of memory errors. For example, if an RDD used by two actions is not persisted, the RDD will be computed twice, and thus degrading the application performance (e.g., SPARK-1266 [3]). If a cached RDD is not used anymore, it occupies much precious memory, and affects the normal execution (e.g., SPARK-18608 [16]). In our experiment, improper caching decisions can seriously degrade the performance of Spark applications (0.1% – 51.6%), and even cause Out of Memory (OOM). In this paper, we refer to improper caching decisions on RDDs as cache-related bugs.

To improve the performance of Spark applications, existing studies mainly focus on improving memory usage in the Spark system, e.g., Yak [39], AstroSpark [45] and Simba [31], and optimizing cache mechanisms in the Spark system, e.g., Neutrino [47], LRC [49] and Lotus [43]. However, cache-related bug patterns and detection approaches in Spark applications have not been studied. Note that, Spark applications have the different execution model from traditional programs (e.g., C/C++ and Java). Thus, the memory/resource leak detection techniques in traditional programs [32, 34, 40] cannot be directly applied on cache-related bug detection in Spark applications.

In this paper, we first analyze the execution semantics of cache mechanisms in Spark, and summarize six bug patterns for cache-related bugs in Spark applications, i.e., `missing persist`, `lagging persist`, `unnecessary persist`, `missing unpersist`, `premature unpersist`, and `logging unpersist` (Section 5). We then propose CacheCheck to automatically detect cache-related bugs in Spark applications. After running a Spark application (without any modification or instrumentation to the application) on our modified Spark system (Section 5), CacheCheck collects the application’s execution trace and caching decisions. Then, CacheCheck infers the correct caching decisions for the application based on the execution semantics of cache mechanisms in Spark. Finally, by analyzing the difference between the original caching decisions and inferred correct caching decisions, CacheCheck reports the inconsistencies as cache-related bugs.

We implement CacheCheck, and make it publicly available at https://github.com/Icysandwich/cachecheck. We evaluate its effectiveness on real-world Spark applications, including GraphX [22] and MLlib [23] and Spark SQL [29], etc. Experimental results show that: (1) CacheCheck can precisely detect all 18 known cache-related bugs in Spark applications. (2) CacheCheck detects 72 previously-unknown cache-related bugs on the latest versions of six Spark applications. We have reported these cache-related bugs to the concerned developers. So far, 53 have been confirmed by developers, of which 28 have been fixed. Developers also express their great interests in CacheCheck.

We summarize the main contributions of this paper as follows.

- We summarize six cache-related bug patterns in Spark applications, which can seriously degrade the performance of Spark applications.
- We propose an automated approach, CacheCheck, to detect cache-related bugs in Spark applications.
- We implement CacheCheck and evaluate it on real-world Spark applications. The experimental results show that it can detect cache-related bugs effectively.

The remainder of this paper is organized as follows. Section 2 explains Spark programming model, cache mechanism, and Spark execution semantics briefly. Section 3 presents the six cache-related bug patterns in Spark applications. Section 4 presents our cache-related bug detection approach. Section 5 presents our CacheCheck implementation, and Section 6 evaluates CacheCheck on real-world Spark applications. Section 7 and Section 8 discuss threats and related work, and Section 9 concludes this paper.

2 BACKGROUND

In this section, we briefly introduce the programming model and the cache mechanism in Apache Spark.

2.1 Spark Programming Model

Spark uses the abstract of Resilient Distributed Dataset (RDD) to store and retrieve data in a distributed cluster. An RDD is an immutable data structure that is divided into multiple partitions, and each of them can be stored in memory or on disk across machines. Developers can perform two types of operations on RDDs: transformation (e.g., `map`) and action (e.g., `count`). Transformations create new RDDs from the existing ones with the specific computation (e.g., user-defined function). Actions return a value (not RDD) by performing a necessary computation (e.g., `count`) on an existing RDD.

A Spark application usually creates RDDs from input data, executes several transformations on RDDs, and performs actions to obtain computation results when necessary [35]. Figure 1a shows a code snippet that contains two jobs: obtaining the total number of words in a text file (Line 4) and taking the first 10 words in the file (Line 5). In this example, we create a new RDD `data` by loading a text file from HDFS (Line 1). Then, we split `data` into words by using transformation `flatMap` (Line 2). To get the total number of words, we perform the action `count on words` (Line 4). To obtain the first 10 words, we perform the action `take on words` (Line 5).

**Lineage graph:** In Spark, all transformations are lazy. When we perform transformations on RDDs, Spark does not immediately perform the transformation computation. Instead, Spark logs all computation dependencies between RDDs in a DAG (Directed Acyclic Graph), which is referred to as lineage graph. Spark keeps on building this lineage graph until an action operation is applied on the last RDD. After that, Spark generates a job to execute each transformation in the lineage graph.

For example, in Figure 1a, `sc.textFile()` (Line 1) and `data.flatMap()` (Line 2) are not executed by Spark immediately. They will only get executed once we perform an action on RDD `words`, i.e., `words.count()` (Line 4). All the transformations in Figure 1a are included in a lineage graph as shown in Figure 1b. The action operation triggers...
1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3: words.persist()
4: words.count()
5: words.take(10)
6: words.unpersist()

(a) Code example

1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3:   words.persist()
4:   words.count()
5:   words.take(10)
6:   words.unpersist()
7:   result.persist()
8:   result.take(10)
9:   result.unpersist()

(b) Prepare A1
(c) Execute A1
(d) Execute A2
(e) After A2

Figure 1: The execution process and cache mechanism in Spark.

Figure 2: Word count example. The left side shows its code, and the right side shows its lineage graph.

the application execution. In other words, the created DAG will be committed to the cluster and the application begins to execute.

2.2 Cache Mechanism in Spark

In the process of an action execution, the transformations that the action depends on are actually computed and the transformed RDDs are materialized. Once the execution completes, these interim RDDs will be released by the Spark system. Therefore, each transformed RDD should be recomputed each time when an action runs on it. As shown in Figure 1a, RDD data and words are used twice by two actions: words.count() (Line 4) and words.take(10) (Line 5). Since each action generates a job, these two RDDs will be computed twice, thus increasing the execution time.

To speed up applications that use certain RDDs multiple times, Spark allows developers to cache data for reuse. By using cache-related APIs, these interim RDDs can be kept in memory (by default) or on disk. So, whenever we invoke another action on the cached RDD, no recomputation takes place. There are two APIs for caching an RDD: cache() and persist(level : StorageLevel). The difference between them is that cache() persists the RDD into memory, whereas persist(level) can persist the RDD in memory or on disk according to the caching strategy specified by level. For easy presentation, we consistently use persist() to represent these two APIs in this paper. Freeing up space from the memory is performed by API unpersist(). In Figure 1a, words is cached by calling persist() on it (Line 3). After the execution of words.count() (Line 4), the data of RDD words is stored in memory. When executing words.take(10) (Line 5), we do not need to recompute data and words anymore. When words is no longer needed, we remove the cached RDD from memory immediately by invoking unpersist() on it (Line 6).

2.3 Spark Execution Semantics

As discussed earlier, when we call a transformation, only the lineage graph is built. When we perform a cache-related API on an reusable RDD, Spark only marks that the RDD should be persisted. The actual caching process takes place when the first action is executed on the RDD. When the next action is executed, Spark finds that the RDD has been cached and thus does not need to recompute it. If this reusable RDD is not cached, recomputation on this RDD takes place and all the RDDs it depends on may be recomputed, too.

Figure 1b shows the dependency relationships between RDDs used by action A1 (Line 4 in Figure 1a) and action A2 (Line 5 in Figure 1a). In Figure 1b, words is marked as persisted (Line 3 in Figure 1a). When A1 is triggered, it tries to find all involved RDDs in the lineage graph from bottom to top. Then the actual execution is performed at the sequence of A1.1, A1.2 and A1.3 in Figure 1c. Since words is marked as persisted, after words is computed, words is stored in the memory (A1.2). When A2 is triggered, it tries to find all involved RDDs first. Since words is already persisted, there is no need to find involved RDDs that words depends on. The actual execution of action A2 starts from words (A2.1 in Figure 1d). After the execution of action A2, words is freed by Line 6 in Figure 1a, as shown in Figure 1e.

3 CACHE-RELATED BUG PATTERNS

In Spark, which RDD should be persisted and when the RDD is persisted / unpersisted are totally decided by developers. However, developers may misunderstand their Spark applications and the caching mechanisms, and introduce cache-related bugs. To combat cache-related bugs, we thoroughly investigate the Spark programming model and cache mechanism. Further, we enumerate all possible ways to utilize cache-related APIs, and evaluate their performance impacts. All cases that can slowdown Spark applications are considered as cache-related bug patterns. Finally, we summarize six cache-related bug patterns.

In this section, we use the word count example shown in Figure 2 to illustrate cache-related bug patterns. RDD data is created by loading a text file from HDFS (Line 1), and further split into words (Line 2) by white spaces. We obtain the number of words by performing action count on words (Line 4). Transformation map maps each word in words with 1 to create a new key-value RDD pairs (Line 5), and transformation reduceByKey further shuffles pairs based on the key and obtains RDD result that contains the number for each particular word (Line 6). We further obtain the total number of unique words in the file by performing action count on result (Line 8), and
obtain the first 10 records in result by performing action take(10) on result (Line 10). Note that, words and result are persisted (Line 3 and Line 7). They are correct decisions for us. Then, we use six bug examples in Figure 3 to illustrate why other persist decisions are incorrect and how they can degrade performance.

3.1 Missing Persist

A transformed RDD in an action will be released by default when the action completes. If an RDD is used by the following actions, it will be recomputed multiple times, thus degrading the execution performance. To improve the execution performance, developers should persist these RDDs that can be used multiple times.

Figure 3a introduces a missing persist bug. In Figure 3a, RDD data and words are used by two actions words.count() (Line 4) and result.count() (Line 8). If words is not persisted, data and words will be computed twice. We do not need to persist data if words is persisted, because if words is persisted, data will not be recomputed. Therefore, the best caching decision is to persist words before the first action (i.e., words.count()) that depends on words is triggered. This bug slows down the example by 21.4%. For another real-world example, the missing persist bug SPARK-16697 [13] in MLlib [23] slows down its application by 51.6% (More details about performance slowdown can be found in Table 1).

3.2 Lagging Persist

As discussed earlier, an RDD is only marked as persisted when the persist() API is invoked. The actual caching process takes place when the RDD is materialized by an action. Therefore, for an RDD that needs to be persisted, the persist() API must be called on the RDD before the first action that depends on it is triggered. Otherwise, the RDD cannot be persisted as expected.

Figure 3b introduces a lagging persist bug. The persist() operation on RDD words (Line 5) is invoked after action words.count() and before action result.count() (Line 9). When action words.count() at Line 4 is executed, words is not marked as persisted, and will not be persisted. When action result.count() is executed, data and words are recomputed again, since words is not cached yet. In this example, lagging persist of words causes recomputation of data and words in action result.count(). This bug slows down the example by 22.9%.

3.3 Unnecessary Persist

Cached RDDs usually occupy large precious memory. Therefore, if an RDD is not reused by multiple actions, it should not be persisted. Unnecessary persist can occupy extra memory, and affects the application execution, thus causing performance degradation.

Figure 3c introduces an unnecessary persist bug on RDD pairs, which is persisted at Line 6. Since RDD result has been persisted after the execution of action result.count() (Line 9). The following action result.take() will use cached result rather than pairs. Thus, it is unnecessary to persist pairs in this example. This bug slows down the example by 4.5%. The real-world unnecessary persist bug SPARK-18608 [16] slows down its application by 23.3%.

3.4 Missing Unpersist

If an RDD is persisted but not unpersisted when it will not be used anymore, the RDD will be kept in memory until the application
An execution trace of a Spark execution can be represented as follows.

\[ \tau = (\text{action}|\text{persist}|\text{unpersist})^\ast \]

In the execution trace, each action contains its partial lineage graph that it directly depends on. For example, in Figure 2, action \( A_1 \) contains its partial lineage graph \( G_1 = \text{data} \rightarrow \text{words} \rightarrow A_1 \), and action \( A_2 \) contains its partial lineage graph \( G_2 = \text{data} \rightarrow \text{words} \rightarrow \text{pairs} \rightarrow \text{result} \rightarrow A_2 \), and action \( A_3 \) contains its partial lineage graph \( G_3 = \text{data} \rightarrow \text{words} \rightarrow \text{pairs} \rightarrow \text{result} \rightarrow A_3 \). Note that, Spark generate a unique ID for each RDD. In the execution trace, the recorded lineage graph is built on these unique IDs. For easy presentation and understanding, we use variable names instead of these unique IDs in the lineage graphs.

We can further combine the partial lineage graphs of all actions to generate the whole lineage graph \( G \). The whole lineage graph generation algorithm is straightforward. For each action \( A_i \), we inspect each edge \( e \) in its lineage graph \( G_i \). If \( e \) does not exist in \( G \), we merge edge \( e \) and its corresponding nodes into \( G \). For example, we can combine \( G_1 \), \( G_2 \), and \( G_3 \), and generate the lineage graph in the right side of Figure 2.

Note that, the execution trace precisely reflects the caching decisions for the Spark application, e.g., which RDDs are persisted, and when they are persisted and unpersisted. Take Figure 2 as an example. After running this application, we can obtain an execution trace as follows: \{words.persist(), words.count(), result.persist(), result.count(), words.unpersist(), result.take(10), result.unpersist()\}. From this trace, we can see that words is persisted before action \( A_1 \) (i.e., words.count()), and words is unpersisted after action \( A_2 \) (i.e., result.count()).

We collect the execution trace using dynamic method by slightly instrumenting Spark’s code. The details are shown in Section 5. Thus, complex code structures, e.g., loops and recursions, will be unfolded as a sequential trace in this step.

### 4.2 Correct Caching Decision Generation

To judge whether the caching decisions in a given actual execution trace \( \tau_a \) are correct or not, we construct the correct execution trace \( \tau_c \), which reflects the correct caching decisions. In order to do so, we extract all actions in the actual execution trace \( \tau_a \) and ignore all caching decisions in \( \tau_a \). Then, we construct the correct execution trace \( \tau_c \) in three steps. First, we identify all RDDs that need to be persisted (Section 4.2.1). Second, we identify the correct locations for persisting these RDDs (Section 4.2.2). Third, we identify the correct locations for unpersisting these RDDs (Section 4.2.3).

#### 4.2.1 Identify RDDs That Need to bePersisted

As discussed in Section 3.1, if an RDD is used by multiple actions, the RDD needs to be persisted. In the lineage graph, the reused RDD is depended by multiple actions. For example, in Figure 2, words is used by action \( A_1 \) and \( A_2 \). Thus, words should be persisted. However, not all RDDs that are depended by multiple actions should be persisted. In Figure 2, pairs is depended by action \( A_2 \) and \( A_3 \). If we persist pairs and do not persist result, RRD result will be computed twice by \( A_2 \) and \( A_3 \), respectively. If we persist pairs and result, the cached pairs will not be reused, and cause an unnecessary persist (Section 3.3). Therefore, CacheCheck needs to identify optimal decisions about what RDDs should be persisted.
Algorithm 1: Identify RDDs that need to be persisted.

Input: actions (Executed action list)
Output: cachedRDDs

1. cachedRDDs ← Φ;
2. foreach (a1, a2) in actions do
   3.   identifyCachedRDDs(a1, rdd, a2);
3. end

Function identifyCachedRDDs(rdd, a2) do

4.   if a2.dependsOn(rdd) then
5.     cachedRDDs.add(rdd);
6.   else
7.     foreach pRDD ∈ rdd.parentRDDs do
8.       identifyCachedRDDs(pRDD, a2);
9.   end
10. end

Figure 4: Identify cached RDDs for the example in Figure 2.

Algorithm 2: Generate correct caching decisions.

Input: actions (Executed action list), cachedRDDs
Output: τc (Correct trace)

1. τc ← Φ;
2. τc.addAll(actions);
3. /* Identify persist locations */
4. foreach rdd ∈ cachedRDDs do
5.   for i ← 1; i ≤ actions.length; i++ do
6.     if actions[i].dependsOn(rdd) then
7.       τc.insertBefore(actions[i], rdd.persist());
8.     break;
9.   end
10. /* Identify unpersist locations */
11. foreach rdd ∈ cachedRDDs do
12.   lastAction ← NULL;
13.   for i ← 1; i ≤ actions.length; i++ do
14.     if use(action[i], actions[i].rdd, rdd) then
15.       lastAction ← actions[i];
16.     end
17.     τc.insertBefore(action[i], rdd.unpersist());
18.   end
19. /* Does curRDD use tarRDD? */
20. Function use(action, curRDD, tarRDD) do
21.   if curRDD = tarRDD then
22.     return True;
23.   else if isCached(action, curRDD) = False then
24.     foreach pRDD ∈ curRDD.parentRDDs do
25.       if use(action, pRDD, tarRDD) then
26.         return True;
27.     end
28.     return False;
29.   end
30. /* Has rdd been cached before executing action? */
31. Function isCached(action, rdd) do
32.   if cachedRDDs.contain(rdd) = False then
33.     return False;
34.     preAction ← τc.previousAction(action);
35.   if preAction ≠ NULL then
36.     if τc.isBefore(rdd.persist(), preAction) then
37.         return True;
38.       return False;
39.   end
40. end

persistence. Algorithm 2 presents how to identify when to persist an RDD. First, we construct an initial execution trace by adding all actions into it (Line 2). For the example shown in Figure 2, the initial execution trace is \{words.count(), result.count(), result.take(10)\}.

For each RDD rdd in cachedRDDs, the operation rdd.persist() should be inserted before the first action that depends on rdd (Line

Algorithm 1 shows how to identify RDDs that need to be persisted for an execution trace. We aim to find the minimal set of RDDs, if they are persisted, all RDD recomputation will be eliminated. For every two actions a1 and a2 in the executed action list actions, CacheCheck finds the closest RDDs that are depended by both a1 and a2 (Line 2–4). CacheCheck starts from the RDD that a1 directly depends on (a1.rdd at Line 3), and checks whether the RDD is used by action a2. If a2 also depends on rdd, then rdd is one of the closest RDDs that are used by two actions, and rdd needs to be persisted (Line 7–8). If a2 does not depend on rdd, CacheCheck further checks whether the parent RDDs of rdd are depended by a2 (Line 10–12).

Take Figure 2 as an example. This example contains three actions: words.count(), result.count() and result.take(10). The cached RDD identification process is shown in Figure 4. For action words.count() and result.count(), the closest RDD shared by them is words as shown in Figure 4a. For action result.count() and result.take(10), the closest RDD shared by them is result as shown in Figure 4b. For action words.count() and result.take(10), the closest RDD shared by them is words as shown in Figure 4c. Therefore, words and result need to be persisted.

4.2.2 Identify Persist Locations. After finding all the RDDs that should be persisted in Algorithm 1, we need to identify when to
Algorithm 3: Detect cache-related bugs.

\textbf{Input:} $\tau_c$ (Correct trace), $\tau_a$ (Actual trace)
\begin{algorithmic}[1]
  \FOR {$i \leftarrow 1; i \leq \tau_a$.length; $i++$}
    \STATE $\op \leftarrow \tau_c[i]$
    \IF {$\op \notin \tau_a$}
      \IF {$\op$ isPersist then}
        \STATE Report 'missing persist on $\op$.rdd';
      \ENDIF
      \IF {$\op$ isUnpersist then}
        \IF {$\op$ isAfter($\tau_a$.lastAction) \&
          $\tau_a$.havePersist($\op$.rdd)}
          \STATE Report 'missing unpersist on $\op$.rdd';
        \ENDIF
      \ENDIF
    \ELSE
      \IF {$\op$ isPersist then}
        \STATE $\cAction \leftarrow \tau_c$.nextAction($\op$);
        \STATE $\aAction \leftarrow \tau_a$.nextAction($\op$);
        \IF {$\cAction$ isBefore($\aAction$)}
          \STATE Report 'lagging persist on $\op$.rdd';
        \ENDIF
      \ELSE if {$\op$ isUnpersist then}
        \STATE $\cAction \leftarrow \tau_c$.previousAction($\op$);
        \STATE $\aAction \leftarrow \tau_a$.previousAction($\op$);
        \IF {$\cAction$ isBefore($\aAction$)}
          \STATE Report 'lagging unpersist on $\op$.rdd';
        \ENDIF
      \ENDIF
      \STATE \textbf{else if} $\aAction$ isBefore($\cAction$)
      \STATE Report 'premature unpersist on $\op$.rdd';
    \ENDIF
  \ENDFOR

4–10). For the example shown in Figure 2, \texttt{words} and \texttt{result} should be cached. For RDD \texttt{words}, \texttt{words.count()} is the first action that depends on it. For RDD \texttt{result}, \texttt{result.count()} is the first action that depends on it. Therefore the new trace that contains the RDD persist locations is: \{\texttt{words.persist()}, \texttt{words.count()}, \texttt{result.persist()}, \texttt{result.count()}, \texttt{result.take}(10)\}.

4.2.3 Identify Unpersist Locations. If a cached RDD is not used any more by following actions, it should be unpersisted as early as possible to save memory. That is to say, once the last action that uses a cached RDD has been executed, the RDD should be discarded. First, we find the last action which uses a cached RDD (Line 13–17). Then, we insert \texttt{rdd.unpersist()} right after the last action (Line 18). To decide if an action uses an RDD \texttt{rdd}, CacheCheck starts to check if the RDD \texttt{curRDD} that \texttt{action} directly depends on uses \texttt{rdd} (Line 15). If \texttt{curRDD} is not cached before, then CacheCheck checks whether its parent RDDs use \texttt{tarRDD} (Line 25–29). If \texttt{curRDD} has been cached (Line 34–42), then CacheCheck will stop search, since \texttt{action} will use cached \texttt{curRDD}, and does not use its parent RDDs. Take Figure 2 as an example. the last action that uses \texttt{words} is \texttt{result.count()} and the last action that uses \texttt{result} is \texttt{result.take}(10). Note that, once \texttt{result} is persisted by action $A_2$, action $A_3$ will use the cached \texttt{result}, and will not use \texttt{words} any more. Therefore, the new correct trace is: \{\texttt{words.persist()}, \texttt{words.count()}, \texttt{result.persist()}, \texttt{result.count()}, \texttt{words.unpersist()}, \texttt{result.take}(10), \texttt{result.unpersist()}\}.

4.3 Bug Detection
CacheCheck detects cache-related bugs by analyzing the difference of caching decisions in the collected actual execution trace ($\tau_a$) and the generated correct execution trace ($\tau_c$) in Section 4.2. For all the buggy examples in Figure 3, CacheCheck can generate the same correct execution trace: \{\texttt{words.persist()}, \texttt{words.count()}, \texttt{result.persist()}, \texttt{result.count()}, \texttt{words.unpersist()}, \texttt{result.take}(10), \texttt{result.unpersist()}\}. However, their actual execution traces are different, and illustrate different cache-related bugs. Algorithm 3 shows our cache-related bug detection. We illustrate it as follows.

(1) Missing persist. If an operation \texttt{rdd.persist()} belongs to the correct execution trace but does not belong to the actual execution trace, CacheCheck reports a missing persist bug (Line 3–5).

(2) Lagging persist. Both the correct execution trace and the actual execution trace contains an operation \texttt{rdd.persist()}. However, in the actual execution trace, \texttt{rdd.persist()} is not performed before the first action that uses \texttt{rdd} (Line 10–14). Take Figure 3b as an example. We can get the actual execution trace as \{\texttt{words.count()}, \texttt{words.persist()}, \texttt{result.persist()}, \texttt{result.count()}, \texttt{words.unpersist()}, \texttt{result.take}(10), \texttt{result.unpersist()}\}. We can see that \texttt{words.persist()} is located before \texttt{words.count()} in the correct trace. However, it is located after \texttt{words.count()} in the actual execution trace. So, CacheCheck reports a lagging persist bug on RDD \texttt{words}.

(3) Unnecessary persist. If an operation \texttt{rdd.persist()} belongs to the actual execution trace but does not belong to the correct execution trace, CacheCheck reports an unnecessary persist bug (Line 25–27).

(4) Missing unpersist. If an RDD \texttt{rdd} that should be cached has been persisted in the actual execution trace, but the operation \texttt{rdd.unpersist()} belongs to the correct execution trace but not belong to the actual execution trace, and the operation is not located after the last action in the trace, CacheCheck reports a missing unpersist bug (Line 6–8).

(5) Premature unpersist. Both the correct execution trace and the actual execution trace contain an operation \texttt{rdd.unpersist()}. However, in the actual execution trace, \texttt{rdd.unpersist()} is performed before the last action that uses \texttt{rdd} (Line 20–21). Take Figure 3e as an example. The actual execution trace of this example is \{\texttt{words.persist()}, \texttt{words.count()}, \texttt{words.persist()}, \texttt{result.persist()}, \texttt{result.count()}, \texttt{result.take}(10), \texttt{result.unpersist()}\}. \texttt{words.unpersist()} is located after \texttt{result.count()} in the correct execution trace, while it is located before \texttt{result.count()} in the actual trace. Thus, CacheCheck reports a premature unpersist bug on RDD \texttt{words}.

(6) Lagging unpersist. Both the correct execution trace and the actual execution trace contain an operation \texttt{rdd.unpersist()}. However, in the actual execution trace, \texttt{rdd.unpersist()} is not performed right after the last action that uses \texttt{rdd} (Line 15–19).

5 IMPLEMENTATION
CacheCheck first collects an execution trace by modifying Spark implementation, and then performs cache-related bug detection offline.
on the execution trace. Here, we mainly introduce the implementa-
tion details which are not described in Section 4, including code
instrumentation for trace collection, and bug reports for helping
inspect bugs and bug deduplication.

**Execution trace collection.** Spark invokes SparkContext.runJob() to execute each action in a Spark application. We modify Spark-
Context.runJob(), and record the information about each action,
including the action ID, the RDD that the action works on, and the
lineage graph of the action. To obtain each cache-related operation,
we modify RDD.persist() and RDD.unpersist(), and record the RDD
ID for each persist() and unpersist() operation. Our modification to
the Spark system is quite lightweight, and involves only 338 lines
of code. The overhead for obtaining execution traces is usually
negligible (less than 5% in our experiments).

Note that, we modify the Spark system to collect execution traces
of Spark applications. Therefore, developers only need to run their
Spark applications on our modified Spark system, without instru-
menting or modifying their applications in any way. CacheCheck
can automatically collect their execution traces and perform offline
analysis for cache-related bug detection.

**Bug reports.** CacheCheck detects cache-related bugs based on
the execution trace. If a cache-related bug at the same code location
is executed multiple times, CacheCheck can report the cache-related
bug multiple times. In order to eliminate duplicate bug reports,
we consider two bug reports are the same if they satisfy the following
two conditions: (1) Their bug-related RDDs are generated in the
same code locations. (2) They have the same bug information, e.g.,
bug pattern and code locations of cache-related API invocations.

To facilitate bug inspection for developers, we provide the fol-
lowing information for each reported bug: (1) The code positions
and call stacks for each operation that relates to the bug report,
e.g., where a related RDD is generated, where a cache-related API
is called on an RDD, and invocation position for each action. (2)
The lineage graph for each action that relates to the bug report.
For example, for a missing persist on rdd, CacheCheck provides
where two actions use rdd, and which action firstly uses rdd. For
a premature unpersist on rdd, CacheCheck provides where the
unpersist() is invoked on rdd, and actions that use rdd after the
unpersist() operation.

6 EVALUATION

We evaluate CacheCheck and study the following three research
questions:

**RQ1:** Can cache-related bugs seriously affect the performance of
Spark applications?

**RQ2:** Can CacheCheck effectively detect known cache-related bugs
in Spark applications?

**RQ3:** Can CacheCheck detect unknown cache-related bugs in real-
world Spark applications?

To answer RQ1 and RQ2, we first build a cache-related bug bench-
mark with 18 bugs (Section 6.1). To answer RQ1, we compare the
performance difference between the buggy versions and their corre-
sponding fixed versions (Section 6.2). To answer RQ2, we evaluate
CacheCheck on our bug benchmark (Section 6.3). To answer RQ3,
we evaluate CacheCheck on six real-world Spark applications to
validate if CacheCheck can detect new bugs (Section 6.4).

### 6.1 Cache-Related Bug Benchmark

We select the six bug examples in Figure 3 as our micro-benchmark
(P1–P6 in Table 1). We further manually inspect issue reports in
open source Spark applications, to collect real-world cache-related
bugs. We select three representative Spark applications, i.e., GraphX
[22], MLlib [23] and Spark SQL [29], as our study applications.
These applications are representative official Spark applications,
well-maintained, and widely used in practice. All issue reports in
these applications are publicly available in Spark JIRA [28].

We use cache-related keywords, i.e., cache and persist, to search
all issue reports in GraphX, MLlib and Spark SQL. Two authors
and two master students carefully analyze each returned issue report.
If an issue report satisfies the following conditions, we select it into
our benchmark. (1) We can identify its root cause, and confirm that
it is a cache-related bug. (2) We can find its fixing patch, and obtain
its buggy version and fixed version. (3) We can reproduce it based
on an existing test case in its application.

Through the above process, we finally obtain 12 cache-related
bugs (B1–B12 in Table 1). Note that, these 12 cache-related bugs
can cause serious performance slowdown: P1, P2, P5, B1, B2, B3,
B5, B8. Specially, in bug B2, all the cached RDDs in a loop are not unpersisted,
and waste huge memory, thus causing Memory Out of (OOM).
After correctly unpersisting all RDDs in the loop, the application terminates normally.

Generally, missing persist (e.g., P1, B5), lagging persist (e.g., P2),
and premature unpersist (e.g., P5) can usually cause much perfor-
ance slowdown, since they can cause recomputation of related
RDDs. However, not all missing persist bugs can cause serious perfor-
ance slowdown, e.g., B6 and B7. This is because recomputing
the unpersisted RDDs in these two cases is not time-consuming.
Missing unpersist (e.g., B2, B3) and unnecessary persist (e.g., B8)
can cause much performance slowdown sometimes, if they occupy
too much memory.

### 6.2 Performance Slowdown Caused by
Cache-Related Bugs

**Experimental design.** We use the 18 cache-related bugs in our
bug benchmark in Table 1 as our experimental subjects. We run this
experiment in the local mode with 8GB memory. For the six cache-
related bugs (P1–P6), we use a 660MB text file as input. For the 12
real-world cache-related bugs (B1–B12), we run their corresponding
test cases with inputs of 23.4–708MB. For each cache-related bug,
we run its buggy version and fixed version 10 times, and then use
the average execution time to compare the performance slowdown.

**Experimental results.** The columns 5–7 in Table 1 (Execution
time) show the absolute execution time of the buggy / fixed versions
and the performance slowdown. We can see that cache-related bugs
may not have realized these three bug patterns before our study.

Generally, missing persist (e.g., P1, B5), lagging persist (e.g., P2),
and premature unpersist (e.g., P5) can usually cause much perfor-
ance slowdown, since they can cause recomputation of related
RDDs. However, not all missing persist bugs can cause serious perfor-
ance slowdown, e.g., B6 and B7. This is because recomputing
the unpersisted RDDs in these two cases is not time-consuming.
Missing unpersist (e.g., B2, B3) and unnecessary persist (e.g., B8)
can cause much performance slowdown sometimes, if they occupy
too much memory.
5.2 Implementation and Evaluation

5.2.1 Dataset and Experiment Design

We use 18 cache-related bugs in Table 1 to evaluate the cache-related bug detection ability of CacheCheck. For each cache-related bug in Table 1, we run its test case, and use CacheCheck to detect cache-related bugs on the collected execution trace.

The last two columns in Table 1 show the detection result. We can see that CacheCheck can detect all 18 cache-related bugs (Detected). CacheCheck further detects 23 new cache-related bugs (New bugs) in these test cases. Two authors and two master students manually inspect these new detected cache-related bugs to validate whether they are real bugs. After manual validation, we find that all these newly detected cache-related bugs are true. Since these new cache-related bugs are detected on old versions of GraphX, MLlib, and Spark SQL, we do not submit them to the developers for further confirmation.

6.3 Detecting Known Cache-Related Bugs

We use the 18 cache-related bugs in Table 1 to evaluate the cache-related bug detection ability of CacheCheck. For each cache-related bug in Table 1, we run its test case, and use CacheCheck to detect cache-related bugs on the collected execution trace.

<table>
<thead>
<tr>
<th>ID</th>
<th>Issue ID</th>
<th>App</th>
<th>Pattern</th>
<th>Execution time</th>
<th>Detected</th>
<th>New bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Buggy(s)</td>
<td>Fixed(s)</td>
<td>Slowdown</td>
</tr>
<tr>
<td>P1</td>
<td>Pattern-MP (Figure 3a)</td>
<td>Word count</td>
<td>Missing persist</td>
<td>31.7</td>
<td>26.1</td>
<td>21.4%</td>
</tr>
<tr>
<td>P2</td>
<td>Pattern-LP (Figure 3b)</td>
<td>Word count</td>
<td>Lagging persist</td>
<td>32.1</td>
<td>26.1</td>
<td>22.9%</td>
</tr>
<tr>
<td>P3</td>
<td>Pattern-UP (Figure 3c)</td>
<td>Word count</td>
<td>Unnecessary persist</td>
<td>27.3</td>
<td>26.1</td>
<td>4.5%</td>
</tr>
<tr>
<td>P4</td>
<td>Pattern-MUP (Figure 3d)</td>
<td>Word count</td>
<td>Missing unpersist</td>
<td>26.3</td>
<td>26.1</td>
<td>0.9%</td>
</tr>
<tr>
<td>P5</td>
<td>Pattern-PUP (Figure 3e)</td>
<td>Word count</td>
<td>Premature unpersist</td>
<td>35.4</td>
<td>26.1</td>
<td>35.7%</td>
</tr>
<tr>
<td>P6</td>
<td>Pattern-LUP (Figure 3f)</td>
<td>Word count</td>
<td>Lagging unpersist</td>
<td>26.2</td>
<td>26.1</td>
<td>0.3%</td>
</tr>
<tr>
<td>B1</td>
<td>SPARK-1266 [3]</td>
<td>MLlib</td>
<td>Unnecessary persist</td>
<td>24.7</td>
<td>23.4</td>
<td>5.2%</td>
</tr>
<tr>
<td>B2</td>
<td>SPARK-3918 [6]</td>
<td>MLlib</td>
<td>Missing unpersist</td>
<td>OOM</td>
<td>27.2</td>
<td>-</td>
</tr>
<tr>
<td>B3</td>
<td>SPARK-7100 [10]</td>
<td>MLlib</td>
<td>Missing unpersist</td>
<td>27.6</td>
<td>26.1</td>
<td>5.9%</td>
</tr>
<tr>
<td>B4</td>
<td>SPARK-10182 [9]</td>
<td>MLlib</td>
<td>Missing unpersist</td>
<td>34.2</td>
<td>33.6</td>
<td>1.8%</td>
</tr>
<tr>
<td>B5</td>
<td>SPARK-16697 [13]</td>
<td>MLlib</td>
<td>Missing unpersist</td>
<td>67.8</td>
<td>44.7</td>
<td>51.6%</td>
</tr>
<tr>
<td>B6</td>
<td>SPARK-16880 [14]</td>
<td>MLlib</td>
<td>Missing persist</td>
<td>262.2</td>
<td>256.3</td>
<td>2.3%</td>
</tr>
<tr>
<td>B7</td>
<td>SPARK-18356 [15]</td>
<td>MLlib</td>
<td>Missing persist</td>
<td>20.9</td>
<td>20.7</td>
<td>0.7%</td>
</tr>
<tr>
<td>B8</td>
<td>SPARK-18608 [16]</td>
<td>MLlib</td>
<td>Unnecessary persist</td>
<td>244.9</td>
<td>198.7</td>
<td>23.3%</td>
</tr>
<tr>
<td>B9</td>
<td>SPARK-26006 [20]</td>
<td>MLlib</td>
<td>Missing unpersist</td>
<td>56.0</td>
<td>54.8</td>
<td>2.0%</td>
</tr>
<tr>
<td>B10</td>
<td>SPARK-2661 [4]</td>
<td>GraphX</td>
<td>Missing unpersist</td>
<td>13.7</td>
<td>13.1</td>
<td>4.6%</td>
</tr>
<tr>
<td>B11</td>
<td>SPARK-3290 [5]</td>
<td>GraphX</td>
<td>Missing unpersist</td>
<td>11.9</td>
<td>11.7</td>
<td>1.8%</td>
</tr>
<tr>
<td>B12</td>
<td>SPARK-7116 [11]</td>
<td>Spark SQL</td>
<td>Unnecessary persist</td>
<td>20.4</td>
<td>20.2</td>
<td>0.1%</td>
</tr>
</tbody>
</table>

6.4 Detecting Unknown Cache-Related Bugs

In this section, we evaluate CacheCheck on six real-world Spark applications, and obtain developers’ feedbacks about our detected cache-related bugs.

6.4.1 Experimental Design. We collect six Spark applications for our experiments from two aspects. First, we use GraphX [22], MLlib [23], Spark SQL [29], which are the three official Spark applications, and well maintained by the Spark community. Second, we collect another three Spark applications from GitHub by the following steps. (1) We used the keyword “Apache Spark” to search GitHub repositories written in Scala, and obtained around 1600 repositories. (2) We sorted these repositories by their stargazers in descending order, and manually checked them one by one. (3) We aimed to select representative data process applications. Thus, If a repository is used for examples, learning, and extensions of Spark framework, we ignored it. (4) If a repository contains test cases, and we can run them on Spark 2.4.3, then we selected it. By following the above steps, we obtained three applications and evaluated CacheCheck on them. The first four columns in Table 2 show the basic information about all these six Spark applications.

We perform our experiments as follows. First, we run the test cases included in these applications on our modified Spark implementation (Section 5) and collect their execution traces. The fifth and sixth columns in Table 2 show the numbers of collected actions and RDDs in the execution traces. Second, we use CacheCheck to detect cache-related bugs based on these execution traces.

In our experiment, a cache-related bug may be triggered multiple times by different test cases. We adopt the approach discussed in Section 5 to remove duplicated bug reports. When counting bugs, we only count unique bugs, i.e., remove duplicated bug reports from different test cases. MCL [8] and t-SNE [7] are built on MLlib. For these two applications, if we detect a bug occurring in MLlib, we only count it once in MLlib, and do not count it in these applications.

6.4.2 Detection Result. Table 2 shows the unique cache-related bugs detected by CacheCheck. In total, CacheCheck detects 72 cache-related bugs, covering the six cache-related bug patterns in Section 3. In detail, CacheCheck detects 34 missing persist bugs (MP), 1 lagging unpersist bug (LP), 18 unnecessary persist bugs (UP), 5 missing unpersist bugs (MUP), 12 lagging unpersist bugs (LUP), and 2 premature unpersist bugs (PUP). We can see that cache-related bugs are common in real-world Spark applications.

We have submitted all these 72 bugs to developers through JIRA and GitHub. So far, 58 bugs have been discussed by developers, and 53 of them have been confirmed as real bugs. The remaining 5 bugs are not decided yet. Developers have fixed 28 bugs. Among the 14 bugs that have not received responses, 9 occur in third-party
applications on GitHub, e.g., MCL [21], mainly due to the inactive maintenance of these applications.

Note that, we have detected 1 lagging persist bugs, 12 lagging unpersist bugs, and 2 premature unpersist bugs in real-world Spark applications. Among them, 11 have been confirmed by developers, and 7 have been fixed. Although we did not observe these bug patterns when we collecting known cache-related bugs, developers confirm that these bug patterns are harmful.

### 6.4.3 Feedbacks from Developers

So far, developers have given feedbacks on 58 cache-related bugs. In the process of bug submission, developers have shown great interests in CacheCheck. For example, after they confirmed and resolved one bug [24], they gave the following comments to encourage us to submit more cache-related bugs: "Nice, I wonder what else CacheCheck will turn up."

Besides, we also received some interesting comments from developers, which indicates that even experienced developers may not be easy to figure out complicated caching decisions. For example, Figure 5 shows the simplified code of our bug report SPARK-29844 [24]. userFactor' denotes a new RDD created in Line 5, which is different from the original RDD userFactor in Line 3.

**Figure 5:** Premature unpersist and lagging unpersist bugs in simplified ALS algorithm in SPARK-29844 [24]. userFactor’ denotes a new RDD created in Line 5, which is different from the original RDD userFactor in Line 3.

6.4.3 Feedbacks from Developers. So far, developers have given feedbacks on 58 cache-related bugs. In the process of bug submission, developers have shown great interests in CacheCheck. For example, after they confirmed and resolved one bug [24], they gave the following comments to encourage us to submit more cache-related bugs: “Nice, I wonder what else CacheCheck will turn up.”

Besides, we also received some interesting comments from developers, which indicates that even experienced developers may not be easy to figure out complicated caching decisions. For example, Figure 5 shows the simplified code of our bug report SPARK-29844 [24] in MLLib. There are two cache-related bugs in this report. (1) itemFactor (Line 8) should be unpersisted after itemOutput.count() (Line 9), instead of after userOutput.count(Line 7). This causes a premature unpersist bug. (2) userInput (Line 10) should be unpersisted before itemOutput.count() (Line 9), instead of after itemOutput.count() (Line 9). This causes a lagging unpersist bug. In this bug, a developer left the following comment in its PR [25]: “I don’t see the problem here immediately. Doesn’t this (itemOutput) depend on some of the cached user RDDs (userInput)?” The developer thought itemOutput depends on userInput, so userInput should be unpersisted after all actions completed. After we explained the fact that itemOutput only depends on the cached itemFactor, he agreed that moving userInput.unpersist() before itemOutput.count() is a kind of optimization.

**Figure 6:** An unnecessary persist bug when Line 4 in the while statement is executed only once in MLLib.

6.4.3 Feedbacks from Developers. So far, developers have given feedbacks on 58 cache-related bugs. In the process of bug submission, developers have shown great interests in CacheCheck. For example, after they confirmed and resolved one bug [24], they gave the following comments to encourage us to submit more cache-related bugs: “Nice, I wonder what else CacheCheck will turn up.”

Besides, we also received some interesting comments from developers, which indicates that even experienced developers may not be easy to figure out complicated caching decisions. For example, Figure 5 shows the simplified code of our bug report SPARK-29844 [24]. userFactor’ denotes a new RDD created in Line 5, which is different from the original RDD userFactor in Line 3.

**Figure 6:** An unnecessary persist bug when Line 4 in the while statement is executed only once in MLLib [26].

(itemOutput) depends on (userInput), so (userInput) should be unpersisted after all actions completed. After we explained the fact that (itemOutput) only depends on the cached (itemFactor), he agreed that moving (userInput).unpersist() before (itemOutput).count() is a kind of optimization.

**Figure 6:** An unnecessary persist bug when Line 4 in the while statement is executed only once in MLLib [26].
1: def apply(vertices, edges): Graph[V, E] = {
2:     vertices.persist()
3:     val newEdges = edges.map(part => part.withoutVertex).persist()
4:     new Graph(vertices, newEdges)
5: }

Figure 7: An API related to cache decisions in GraphX [17].

examples), and developers thought fixing the bugs can make code understanding difficult, e.g., SPARK-29872 [27].

(2) Developers thought that fixing cache-related bugs makes APIs difficult to utilize. 8 bugs belong to this case. Spark applications, e.g., GraphX, MLLib and Spark SQL, can be used as APIs to build other applications. The caching decisions of these APIs may not consider all their usage scenarios. Figure 7 shows an unnecessary persist bug in GraphX [17]. In API apply(), RDD vertices and newEdges are persisted. This API can be invoked by many graph processing algorithms in GraphX, e.g., SVDPlusPlus and PageRank. Both vertices and newEdges are used by only one action in SVD-PlusPlus, so CacheCheck reports two unnecessary persist bugs on them. However, in PageRank, they are used by multiple actions in an iterative computation, so the persist decisions are correct. Developers wanted to simplify the usage of these APIs, and would not like to fix them for different applications.

6.4.4 Slowdowns of Detected Unknown Bugs. The slowdowns of cache-related bugs are usually highly related to the size of input data in the Spark applications. We detected unknown cache-related bugs by running test cases included in the applications. Note that, these test cases usually run on small amount of data (e.g., a string with 100 characters, a graph with 10 nodes), and complete quickly (e.g., in less than 1 second). In these scenarios, the slowdowns can be ignored. Therefore, we did not report the slowdowns for these test cases. However, for real scenarios, these test cases will be run on large amount of data (e.g., 100G) for a long time (e.g., 30 minutes). In these scenarios, the slowdowns will be greatly important. Thus, for the newly reported bugs that have been fixed, we run their test cases with large amount of data like what we did in Section 6.2, and analyzed their slowdowns. Our experiment shows that, the slowdowns range from 0.2% to 35.4%, and 3 bugs cause Out of Memory (OOM).

7 DISCUSSION

While our experiments show that CacheCheck is promising in detecting cache-related bugs in Spark applications, we discuss potential threats and limitations of our work.

Representativeness of our studied Spark applications. We select a number of cache-related bugs and Spark applications in our experiments. Our selected cache-related bugs come from real-world Spark applications. Our selected Spark applications, e.g., GraphX [22], MLLib [23], and Spark SQL [29], are well maintained and actively developed by the Spark community. Thus, we believe our studied cache-related bugs and Spark applications can represent the real-world cases.

Trade-off between memory and performance. Whether an RDD should be persisted or not is usually a trade-off between execution performance and the memory requirement of the persisted RDD. Our approach does not take the memory requirement into consideration. However, we believe that CacheCheck can still give developers useful suggestions about how to enforce proper caching decisions. Additionally, Spark provides various cache storage levels, e.g., off-heap, on-disk, in-memory. CacheCheck cannot provide useful suggestions for cache levels.

Generalization to other systems. CacheCheck is currently designed for Spark. It may be generalized to other big data systems that have the same or similar cache mechanisms as Spark. For example, in order to reduce the cost of recomputation and reloading files, Hadoop and Flink have utilized or planned to utilize similar cache mechanisms. CacheCheck may be adapted into these systems.

8 RELATED WORK

To the best of our knowledge, no previous work can detect cache-related bugs in Spark applications. In this section, we discuss related work that is close to ours.

Cache optimization in Spark. Existing work mainly focuses on cache optimization techniques in Spark implementation. Neutrino [47] employs fine-grained memory caching of RDD partitions and uses different in-memory cache levels based on runtime characteristics. Yang et al. [48] developed some adaptive caching algorithms to make online caching decisions with optimality guarantees. Zhang et al. [52] studied the influence of disk use in the Spark cache mechanism, and proposed a method of combined use of memory and disk caching. These works may eliminate the use of persist() and unpersist() operations and cache levels. To meet this all-or-nothing property in data access, PACMan [30] coordinates access to the distributed caches and designs new cache replacement mechanisms. LRC (Least Reference Count) [49] exploits the application-specific DAG information to optimize the cache replacement mechanism. However, it is still a common practice to enforce caching decisions by developers causing severe performance slowdown. CacheCheck automatically detects improper persist() and unpersist() usage in Spark applications. Thus, our CacheCheck is orthogonal to these pieces of existing work.

Memory management optimization in Spark. Researchers have proposed some approaches to improve memory management in big data systems. An experimental study [46] has shown that garbage collectors can affect big data processing performance. Yak [39] divides the managed heap in JVM into a control space and a data space, and optimizes garbage collection for big data systems. Skyway [38] can directly connect managed heaps of different (local or remote) JVM processes, and optimizes object serialization / deserialization. Stark [36] optimizes in-memory computing on dynamic collections. Panthera [42] analyzes user programs running on Apache Spark to infer their coarse-grained access patterns, and then provides fully automated memory management technique over hybrid memories. Different from these approaches, CacheCheck focuses on the caching decisions in Spark applications.

Resource leak detection in traditional programs. Resource leak in programs can cause severe problems, e.g. performance degradation and system crash [44]. Researchers have made lots of efforts to detect resource leak. Sigmund et al. [32] and Yulei et al. [40] utilized value-flow analysis to detecting memory leaks in C programs. Emina et al. [41] developed Tracker to detect resource leak in Java.
programs. Relda [34] and Relda2 [44] detect resource leak in Android applications. However, all these works cannot handle with the execution model of Spark, e.g., lazy execution. Thus, they cannot be applied to detect cache-related bugs in Spark applications.

9 CONCLUSION

Apache Spark uses caching mechanisms to reduce the RDD recomputation in Spark applications, and relies on developers to enforce correct caching decisions. In this paper, we summarize six cache-related bug patterns in Spark applications. Our experiments on real-world cache-related bugs show that cache-related bugs can seriously degrade the performance of Spark applications. We further propose CacheCheck, which can automatically detect cache-related bugs by analyzing the execution traces of Spark applications. Our evaluation on real-world Spark applications shows that CacheCheck can effectively detect cache-related bugs. In the future, we plan to further improve the detection capability of CacheCheck using new techniques, e.g., static analysis, and performance estimation of cache-related bugs.

ACKNOWLEDGEMENTS

We thank Kai Zhang, Yangfe Fang, Kai Kang, and Xingtong Ye for their contributions in validating detected unknown bugs. This work was partially supported by National Key Research and Development Program of China (2017YFB1001804), National Natural Science Foundation of China (61732019, 61802377, 61702490), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences.

REFERENCES


