Flint: Fixing Linearizability Violations

Peng Liu
Purdue University
peng74@cs.purdue.edu

Omer Tripp
IBM Research, NY
otripp@us.ibm.com

Xiangyu Zhang
Purdue University
xyzhang@cs.purdue.edu

Abstract
Writing concurrent software while achieving both correctness and efficiency is a grand challenge. To facilitate this task, concurrent data structures have been introduced into the standard library of popular languages like Java and C#. Unfortunately, while the operations exposed by concurrent data structures are atomic (or linearizable), compositions of these operations are not necessarily atomic. Recent studies have found many erroneous implementations of composed concurrent operations.

We address the problem of fixing nonlinearizable composed operations such that they behave atomically. We introduce Flint, an automated fixing algorithm for composed Map operations. Flint accepts as input a composed operation suffering from atomicity violations. Its output, if fixing succeeds, is a composed operation that behaves equivalently to the original operation in sequential runs and is guaranteed to be atomic. To our knowledge, Flint is the first general algorithm for fixing incorrect concurrent compositions.

We have evaluated Flint on 48 incorrect compositions from 27 popular applications, including Tomcat and MyFaces. The results are highly encouraging: Flint is able to correct 96% of the methods, and the fixed version is often the same as the fix by an expert programmer and as efficient as the original code.

1. Introduction
Writing concurrent programs is a challenging task. The main challenge is to enable a high level of concurrency (i.e., thread interleavings) while at the same time ensuring that interleaved execution does not lead to unexpected program behaviors, such as data races or deadlocks. Most developers do not have special expertise in parallel programming. Hardware architectures are, however, becomingly increasingly more parallel, and so the demand for efficient yet correct parallel software is on the rise [29]. To address this tension, popular programming languages like Java and C# have incorporated concurrent data types into their libraries.

The System.Collections.Concurrent namespace in .NET offers concurrent implementations of abstract data types (ADTs) frequently in use, like Dictionary and Stack. In Java, the java.util.concurrent package provides similar support with concurrent data structures such as ConcurrentHashMap and BlockingQueue. These container classes, implemented by expert programmers, encapsulate all the complexities entailed by correct yet efficient synchronization, permitting the developer to work with convenient interfaces like putIfAbsent in ConcurrentHashMap or AddOrUpdate in ConcurrentDictionary. These guarantees atomic, or linearizable, behavior. That is, the operation is guaranteed to take effect instantaneously at some point between its invocation and return [3][11].

Composed Concurrent Operations While library support obviates many potential errors and inefficiencies, a fundamental problem still remains: Custom composition of atomic operations does not necessarily render the composed operation atomic. As an illustrative example, we refer to the real-world composed operation in Figure 1. The first step of this operation is to check whether the shared queues Map has a mapping for key sender. If so, then that value is returned. Otherwise, a fresh value is created and mapped under sender assuming another thread hasn’t already created a mapping for sender. If the mapping succeeds, i.e., the putIfAbsent operation returns null, the returned value dests is the fresh value just mapped by the current thread; If the mapping fails, i.e., the putIfAbsent operation returns a non-null value, the returned value dests, obtained via a second get call, is the one already mapped under sender by another thread.

Even though the individual operations comprising put are all atomic, the behavior of put is not atomic. The following concurrent execution scenario, which assumes that in the initial state there is no mapping for sender under queues, demonstrates this:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>get(sender) / sender\rightarrow null</td>
<td>remove(sender)</td>
</tr>
<tr>
<td>putIfAbsent(sender, dests) / succeeds</td>
<td></td>
</tr>
<tr>
<td>get(sender) / sender\rightarrow null</td>
<td></td>
</tr>
</tbody>
</table>
Following a linearization of the `put` operation and the `remove` operation wherein `put` is linearized before `remove`, the `put` operation should either return the value already mapped to `key` or create a fresh mapping and return the respective value. Following the other linearization, wherein `remove` is linearized before `put`, the `put` operation should create a fresh mapping and return the respective value. In either linearization, `put` should not return `null`, though this occurs in the concurrent run above. This shows that the operation is not linearizable.

Oversight of the lack of composability of concurrent operations is a common source of errors. In a recent study, Shacham et al. [27] have found 56 distinct atomicity violations in a large set of real-world concurrent Java programs, including Apache Tomcat, Trinidad and MyFaces. All of these violations, without exception, are due to incorrect composition of atomic `Map` operations. Other studies report similarly alarming statistics [18, 19].

**Scope and Approach** We present a general approach for fixing incorrect (i.e., nonlinearizable) compositions. Similarly to recent studies [18, 26, 28], we instantiate our approach over composed `Map` operations [26]. `Map` is the most popular Java ADT. `Map` compositions are prevalent yet often buggy. We stress that the underlying principles and techniques are of general applicability and can be extended to other data structures implementing ADTs [9].

Our approach breaks into three main steps:

1. **Bottom-up specification extraction:** The first step is to lift the concrete concurrent operation to the level of a semantic description of its intended behavior. This is done by applying a data-flow analysis over the semantic effects of individual operations assuming sequential execution of the composed operation.

2. **Top-down implementation:** Next, based on the specification of how the operation should behave, we reimplement it via a synthesis algorithm geared toward minimizing the number of individual `Map` operations. The rationale underlying minimization is to reduce interleavings between `Map` operations as much as possible. At the extreme, reimplementation reduces the composed operation to a single `Map` operation, which naturally guarantees linearity. A common composition pattern of this kind is “check-then-act” [13]. The composed operation first runs a test and then conditionally performs an update operation (e.g., checking whether the `Map` contains a certain key/value pair and only then removing it). Such compositions often lead to buggy behaviors when the check and the update are not atomic. Minimization is effective in restoring correctness via the built-in check-then-act operations like `putIfAbsent`.

3. **Verification and optimization:** For the resulting minimized version, we verify whether it is linearizable by applying an off-the-shelf linearizability checker [26, 28]. In certain failure scenarios, correctness is recovered by introducing a limited measure of speculation. On the other hand, the verified operations undergo correctness-preserving performance optimizations to avoid unnecessary computations when possible.

We have implemented a prototype of our approach, dubbed Flint. Flint was designed to serve as an IDE refactoring tool, where responsiveness is key. To meet this requirement, Flint leverages a recent result in linearizability checking [26, 28], whereby full verification is replaced by lightweight checking under a criterion (or restriction) known as *data independence*. Intuitively, data independence asserts that the behavior of the composed operation is independent of the actual values of the arguments it operates on, e.g., the operation decides whether to insert a key/value pair based only on membership checking, and not the actual value of the key. The linearizability checker verifies data independence as a precondition, thereby simplifying the ensuing verification process. (See Section 2.5.)

In our practical experience, composed `Map` operations are often data independent. Indeed, in an empirical study we conducted over 48 incorrect compositions from 27 popular applications, we discovered that 38 compositions, which fall into 12 different buggy patterns (see Figure 14), are either data independent or can become data independent following a simple transformation. (See Section 5.2.) Flint is also able to address certain forms of data-dependent composition by inserting abort/retry logic into the transformed code. (See Section 3.2.) Overall, Flint is able to fix 96% of the cases, fixing time is negligible (under 100 milliseconds), and the resulting implementation is mostly comparable to the original buggy code in performance. These suggest that Flint can serve as an effective refactoring tool within IDEs like Eclipse. We point out limitations of Flint in Section 6.

**Contributions** This paper makes the following principal contributions:

- **General fixing approach:** We present a general technique for fixing atomicity violations (or linearizability violations) in composed concurrent operations, which combines bottom-up abstraction of the operation (Section 3) with top-down reimplementation of its behavior (Section 4).

- **Effective fixing strategy:** We demonstrate, as part of our fixing algorithm, that reimplementing the buggy operation using a minimal number of atomic operations is an efficient and robust fixing strategy (Section 4.1). We formalize this strategy as a variant of the well-known setback problem, which we solve as a linear-programming problem. We describe two enhancements of the core strategy: (1) the performance optimization and (2) the correctness recovery via rollbacks (Section 5).

- **Experimental evaluation:** We have evaluated our approach on a set of 48 real-world concurrent operations that suffer from atomicity violations (Section 7). We report on highly encouraging results. We include concrete
compute(k) {
  v = m.get(k);
  if (v != null) {
    return 2*v;
  } else {
    newV = computeV();
    m.putIfAbsent(k,newV);
    return m.get(k);
  }
}

Figure 2. Nonlinearizable composed operation from Adobe BlazeDS

elements of fixes, which illustrate the general ability by our approach to arrive at a transformed version that is both efficient and natural. We have also organized the bugs we encountered into patterns, which may benefit future research.

2. Technical Overview

To illustrate and motivate our technical approach, we refer to the compute composed operation in Figure 2, taken from the Adobe BlazeDS application. In a sequential setting, this method first checks if the shared Map m contains a mapping for the argument key k. If so, then the respective integer value is first doubled and then returned. Otherwise, compute attempts to establish a mapping for k, which would succeed in a sequential run. The returned value in this case is the new value, newV, defined via a call to compute newV, which is obtained by invoking get. For our discussion, we assume that computeV is pure and returns a non-null result.

As we soon demonstrate (Section 2.1), compute suffers from atomicity violations. Flint addresses these violations by abstracting compute into a semantic description of its behavior (Section 2.3) and then concretizing the semantic behavior into another implementation (Section 2.4) that is either guaranteed to be linearizable by construction or verified as linearizable using an off-the-shelf checker. Therefore, Flint either returns an alternative implementation of the original method, such that (i) the new implementation is equivalent to the original method in the sequential setting and (ii) the new implementation is linearizable, or it reports a failure.

2.1 Correctness Criterion: Linearizability

In the context of linearizability, an operation starts with invocation and ends with response. An operation may overlap with another operation, e.g., its response is after the invocation of the other operation while its invocation is earlier. Linearizability [8,11] (or atomicity) is a correctness criterion for concurrent objects that, stated intuitively, provides the illusion that an operation applied to the object takes effect instantaneously at some point between its invocation and its response, known as the linearization point. A concurrent object is considered linearizable if any execution of its operations is equivalent to a “legal” sequential execution that preserves the order between non-overlapping operations. Here the sequential execution linearizes the operations so that none of them overlap. Formal definitions appear in Appendix A. Researchers also refer to linearizability as atomicity in general [27].

We illustrate the meaning of linearizability via our running example. The method compute manipulates a shared ConcurrentHashMap object via its atomic APIs get and putIfAbsent. However, compute is not atomic. As one example, consider a history where initially there is no mapping for key k, so the thread t executing compute transitions into the else branch. Another thread then interleaves an invocation of remove(k) after the call to putIfAbsent (line 8) and before the call to get (line 9). The return value is then null, which is impossible in any legal sequential execution that linearizes the invocations of the composed method and the remove method.

2.2 Fixes

A first attempt to address this atomicity violation, illustrated in Figure 3, is to remove the second get call. This is however insufficient, because another thread may insert pair (k, v'), such that v' != newV, into m in between the calls to get and putIfAbsent. Then the newV result of compute would not even be mapped to k under m. The second attempt, appearing in Figure 4, which ensures that the value x that k is actually mapped to is returned, is also wrong, because the sequential specification dictates that 2 * x should be returned if x is already mapped to k (line 4).

Given this observation, we arrive at the fix in Figure 5, which is correct but may cause performance degradation, because the method computeV is called directly, without first checking the necessity of doing so, i.e., whether k is already mapped under m. We resolve this issue in the final fix presented in Figure 6 which is also computed by Flint. It optimizes the fix in Figure 5 by first checking if k is already in m. In the following, we give an informal account of how Flint arrives at this fix.

2.3 Abstract Behavior

The first step is to derive the specification of the semantic behavior of the operation from its concrete implementation. The semantic behavior of an operation assumes sequential execution. It consists of accesses to, and manipulations of, shared data structures at the level of the ADTs they represent. As an ADT, a Map is an associative container mapping keys to values. As such, its ADT representation is as a (partial) function from keys to values. Map operations, like put and get, manipulate the ADT representation as a function. Flint computes the abstract behavior of composed operations by (i) translating Map operations into their semantic meaning (e.g. v = m.get(k) becoming v = m(k)) and then (ii) computing the resulting semantic expressions over local variables by applying a data-flow analysis, e.g., the analysis computes the semantic expression u = 2*v. The full technical details appear in Section 3.

1 In this paper, we focus on composed operations over Maps.
Figure 3. Incorrect fix following the pattern in the Annsor application

Figure 4. Incorrect fix following the pattern in the autotodroid application

Figure 5. Correct yet inefficient fix

Figure 6. Correct and efficient fix

Figure 7. Abstract (sequential) flow of the operation in Figure 2

2.5 Linearizability Checking

Given an abstract specification of the (sequential) behavior of the composed operation, the goal of the reimplementation step is to realize the specification as a linearizable implementation. Our synthesis algorithm searches for an implementation that consists of a minimal number of Map operations. The underlying rationale is that minimizing the number of Map operations reduces the threat of unexpected thread interleavings. In practice, the minimization often leads to the implementation that consists of only one Map operation, which is linearizable by definition.

To illustrate this strategy, we refer to the graph in Figure 7. There are two possibilities for implementing the \( k \in m \) test. One is to use `containsKey`, and the other is via a `get` call, of which the return value is checked for nullness. Our minimization strategy biases the synthesis algorithm toward get, because the call to `get` achieves both (i) nullness checking and (ii) the value mapped to key \( k \), if such a mapping exists, to implement the “return: \( m(k) \)” node. Using `containsKey` instead would require a separate call to get (or another API).

To achieve minimality, we model the challenge of which operations to select for the (re)implementation as an optimization problem, which we solve using an off-the-shelf linear-programming solver. Given the abstract specification of the composed method, our synthesis algorithm first establishes a set of concrete operations that are candidates to implement different portions of the abstract description, and then minimizes the overall number of operations required to implement the semantic specification. Minimality creates a bias toward correctness. When only one Map operation is used, linearizability is guaranteed. Interestingly, this occurs often. 79% of the buggy real-world compositions we studied can be reduced to a single Map operation. According to our empirical findings, the composed method typically involves a check and then an action over the Map state, which can lead to buggy behaviors when the check and the action are not atomic. (See e.g. Figure 3.)
2.6 Enhancements

Having arrived at a minimal (re)implementation that is verified by the linearizability checker, Flint performs correctness-preserving post-processing optimizations to recover performance, since minimality may degrade performance although it creates a bias toward correctness. Following the common fast-path idiom, the optimization stage attempts to boost the minimal implementation by fusing into it the tests that guard against redundant computations. For instance, given the minimal reimplementation in Figure 5, the optimized version of it, as illustrated in Figure 6, would condition its execution on a check whether \( m\.get(k) \) is null. The optimized code is not minimal, but is more efficient than the minimal reimplementation, and is also proven to be linearizable (Section 5.1).

In the cases where the revised implementation fails linearizability checking, to boost the completeness of Flint, we have developed a fallback strategy that attempts to impose linearizability on the reimplemented code via lightweight rollback facilities. These are limited to rolling back local though not global state manipulated by the execution prefix. In our evaluation, we demonstrate that with this fallback strategy in place, Flint is able to address 96% of the bugs in our input suite compared to 79% otherwise. Note that the rollback strategy complements Flint, but cannot replace it. Concretely, 23 out of the 48 bugs that we examined can be fixed by Flint but cannot be fixed by rolling back local state.

2.7 Discussion

To summarize, Flint first extracts a summary of the intended behavior of the buggy operation, derived from its sequential flow; it then reimplements the operation while restricting the number of Map invocations to a minimum, later applying correctness-preserving performance optimizations; finally, lightweight linearizability checking is performed, and abort/retry logic is installed if needed and possible. All of the above steps aim for efficiency to enable Flint as an IDE refactoring tool.

A natural alternative to Flint is to obtain a fix via code mutation. While possible in theory, this approach would require the automated tool to traverse a vast space of possible mutations, hindering performance and thus also IDE compatibility. Moreover, synthesis is not guaranteed to yield a linearizable fix, as the examples in Figures 3 and 4 illustrate. Flint, instead of attempting arbitrary mutations and transformations, is guided by the intended behavior of the buggy method and minimizes interference with other threads by restricting the number of Map operations to a minimum. According to our experimental results, these are highly effective strategies. (See Section 7.3.)

3. Abstraction

The purpose of abstraction is to specify the behavior (i.e., flow and functionality) of the method at the semantic level, so as to guide the reimplementation process. Since concurrent building blocks are often data structures that implement ADTs, a useful form of abstraction is in terms of ADT semantics. For the Map data structure, a natural abstraction is a (partial) function from keys to values \( \text{Map} \), which is manipulated via the following three abstract operations:

\[ k \in m \text{ indicates whether key } k \text{ is mapped under } \text{Map} m. \text{ We also interchangeably use } m(k) \neq \text{null}, \text{ which is equivalent according to the semantics of } \text{ConcurrentMap}. \]

\[ u = m(k) \text{ denotes access to the value } v \text{ mapped to key } k \text{ under Map } m. \text{ That value is stored as } u. \]

\[ m[k \mapsto v] \text{ sets the respective value of key } k \text{ under } m \text{ to } v. \]

### Table 2. Kill/gen rules for fixpoint computation of available semantic expressions (\( \ast \) stands for any variable or map value.)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Kill</th>
<th>Gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u=m(k) )</td>
<td>( u \mapsto \ast )</td>
<td>( u \mapsto m(k) )</td>
</tr>
<tr>
<td>( u=v )</td>
<td>( u \mapsto \ast )</td>
<td>( u \mapsto [v] )</td>
</tr>
<tr>
<td>( m[k \mapsto v] )</td>
<td>( \ast \mapsto m(k) )</td>
<td>( v \mapsto m(k), k \in m )</td>
</tr>
<tr>
<td>( m[k \mapsto null] )</td>
<td>( \ast \mapsto m(k) )</td>
<td>( k \notin m )</td>
</tr>
</tbody>
</table>

### Figure 8. First step: abstraction of Map operations

```java
compute(k) {
    v=m(k);
    if (v!=null) {
        // \( \{[k \in m, v \mapsto m(k)], [k \notin m, v \mapsto m(k)]\} \)
        return 2*v;
    } else {
        // \( \{[k \in m, v \mapsto m(k)]\} \)
        return m[k];
    }
}
```

### Figure 9. Second step: rewriting of local expressions

```
compute(k) {
    v=m(k);
    newV=computeV();
    if (m(k) == null) {
        return 2 \cdot m(k); }
    else {
        m[k \mapsto newV];
        m[k \mapsto computeV()];
        return m(k);
    }
```
Abstraction

3.2 Abstracting Local Expressions

The next step, having rewritten the operations, is to rewrite local expressions into their semantic meaning. This is important for two reasons: (1) Rewriting recovers missed instances of data independence, a precondition for ensuring linearizability. As data independence requires the method’s result to depend on the Map state, we rewrite the local variable as the Map state it represents. (2) Some local variables represent the Map state in all sequential settings but not in (all) concurrent settings. The programmer, who is not aware of the difference, may use local variables as an optimization for getting rid of a get call, leading to incorrect behaviors in concurrent settings. We need to recover the original semantics of the local variable by replacing it with the Map state that it is expected to represent by both settings.

An illustration of the result of abstracting local expressions is given in Figure 9. As an example, return $2*v$ becomes return $2 \cdot m(k)$ according to this rewriting step, which highlights how to compute the return value (along the relevant branch) in the rewritten method.

To assign semantic expressions to local variables, we perform intraprocedural data-flow analysis that takes its inspiration from the available-expressions analysis [11]. Essentially, our analysis forward propagates, using the classic kill/gen framework [14], the “must” information about the availability of semantic expressions. As a simple example, consider the following code:

```
x = get(k); y = 2*x;
```

Due to operation rewriting, we conclude that $x$ carries the value of expression $m(k)$. Therefore, $y$ maps to the expression $2 \cdot m(k)$.

The data-flow equation underlying our kill/gen available-expressions analysis first applies kill and only then gen:

\[
\text{avail}(n) = \begin{cases} 
\emptyset & \text{if } \text{pred}(n) = \emptyset \\
\bigcap_{p \in \text{pred}(n)} (\text{avail}(p) \setminus \text{kill}(p)) \cup \text{gen}(p) & \text{o/w}
\end{cases}
\]

The data-flow analysis is applied to the Static Single Assignment (SSA) [8] representation. We apply this equation to the nodes of the control-flow graph of the subject method until a fixpoint is reached. The invocation of the methods (non-Map methods) other than Map APIs are assumed to be pure, i.e., the only effect of them are the return values. In the presence of the non-Map methods that are not pure, linearizability is related not only to the Map structure but also to the shared data structures modified by the non-Map methods. As we focus on the linearizability of composed Map operations, we assume the purity of the non-Map methods. The fixpoint process commences after we have applied operation rewriting. The resulting data-flow annotations reflect the semantic behavior of the method assuming sequential execution.

Table 1 lists the kill/gen rules defining the analysis. A read operation sets the assigned variable to point to the read expression $m(k)$. Assignment between variables kills the mapping previously established for the left-hand variable $u$, and instead points it to the target of the right-hand variable.

As an illustration, consider the real-world code in Figure 10, taken from the Annsor application. The local variable $c$ in the return always represents a Map state in sequential runs. However, in the concurrent settings, if the execution of get+key is interleaved just before line 5 by the insertion of pair (a,v) (for some value $v \neq c$) by another thread, the local variable $c$ does not represent a Map state. The programmers, who bear the sequential reasoning in mind, use the local variable as the
replacement of the Map state, leading to incorrect behaviors in concurrent settings.

Our analysis recovers the original semantics of such local variables by applying the data flow analysis \(^2\) under the sequential reasoning. As illustrated in Figure 10, there are two possibilities after line 2: (1) \(a \in m, c \mapsto m(a)\) or (2) \(a \notin m, c \mapsto m(a)\). Only the second data flow can be propagated through the if branch, which changes to \(a \in m, c \mapsto m(a)\) at the end of the branch. Combining the data flows following both the if branch and else branch at line 11, we know the variable \(c\) is equivalent to the map state \(m(a)\) and therefore abstract it as the map state.

In addition, abstracting the local variable in the return, as demonstrated above, also recovers the data independence, which is a prerequisite for a composed operation to be verified as linearizable. Specifically, data independence requires the return depends only on the Map state (Section 2.5), which may however be violated. For example, the return variable \(c\) in Figure 10 may not represent any Map state in concurrent settings. In contrast, the return in Figure 11 always depends on the Map state \(m(a)\).

4. Reimplementation

Having performed the bottom-up mapping from the original code to its semantic specification describing the abstract behavior, our goal in the synthesis step is top-down projection of the semantic specification into a concrete implementation. As an example, we would like to synthesize the code in Figure 5 or Figure 6 based on the semantic description in Figure 7 or Figure 9. Synthesis essentially replaces the abstract operations in the specification with concrete operations.

If abstract operation \(a\) can be implemented by concrete operation \(c\) (i.e., \(c\) produces the expression(s) in \(a\)), then we say \(c\) covers \(a\). A may be covered by multiple candidate concrete operations, collectively denoted as \(\text{covered}(a)\), besides, \(c\) may cover multiple abstract operations, collectively denoted as \(\text{covered}(c)\).

Given the abstract operations (or more precisely, instantiations of the abstract operations) from the semantic specification, \(a_1, \ldots, a_m\), we need to first determine the candidate concrete operations that cover them, \(c_1, \ldots, c_n\), and then select a minimal set of concrete operations out of the candidates to cover all the abstract operations. We defer the explanation of how we arrive at the candidate concrete operations to Section 4.2. For now, we assume that we have all the candidate concrete operations. Our motivation for minimizing the number of operations is that there are fewer interleavings and the linearizability is more likely to be obtained. See also Section 5 for additional discussion and formal claims.

4.1 Covering Operations

The set-cover problem is the following combinatorial problem: Given the set \(U\) of elements (known as the universe) and the family \(S = \{S_1, \ldots, S_n\}\) of subsets \(S_i\) of \(U\) such that \(\bigcup S_i = U\), the goal is to find a cover—i.e., a subset \(\{S_{i_1}, \ldots, S_{i_k}\}\) of \(S\), such that \(\bigcup_{1 \leq j \leq k} S_{i_j} = U\)—whose cardinality is minimal. We strengthen the constraints, further requiring that any given element in \(U\) is covered by exactly one set \(S_i \in S\), which yields a slight variant of the original problem.

In our setting, there are candidate concrete Map operations and the abstract operations. We select the minimal set of concrete operations to cover the abstract operations by solving the cover problem, in which each concrete operation \(c_i\) is treated as a set of abstract operations that it covers, analogously to \(S_i\) in the above. For example, given the fragment \(k \in m \? \text{return } m(k) : \text{return } \text{null}\), the concrete operation \(\text{v=get}(k)\) covers both abstract Map operations, \(k \in m\) and \(\text{return } m(k)\), where \(v\) can be checked for nullness to cover \(k \in m\), and if \(v\) is not null, then it stores the expression \(m(k)\).

We then solve the following linear-programming problem:

\[
\min \sum_{c_j \in C} \sum_{a_i \in \text{covered}(c_j)} x_{ij} \quad (1)
\]

\[
\forall a_i \in A, \sum_{c_j \in \text{covered}(a_i)} x_{ij} = 1 \quad (2)
\]

Boolean variable \(x_{ij}\) denotes whether the expression \(a_i\) is covered by the operation \(c_j\) \((x_{ij} = 1)\). Equation 2 specifies that every abstract operation is covered by exactly one concrete operation. Equation 1 enforces that a minimal set of concrete operations should be used. Specifically, an invocation \(c_j\) is used if and only if it covers at least one abstract operation: \(\sum_{a_i \in \text{covered}(c_j)} x_{ij} = 1\). We solve the linear program by invoking a standard solver. (See Section 7)

In addition, we must account for control dependencies. Separating abstract operations that are control dependent into two (or more) concrete operations typically introduces harmful interleavings \(^19\). Therefore, we impose an additional constraint that such abstract operations should be covered by a single concrete operation. As an example, if the

---

\(^2\) The data flow analysis is applied to the SSA form, but we demonstrate with the non-SSA form for simplicity.
operation \( c \) installed at node \( n \) whether the expressions associated with adjacent nodes are also covered by \( c \). If they are also covered by \( c \), then we include them into \( \text{covered}(c) \), i.e., the set of abstract operations covered by \( c \), which is the output of this step.

As an illustration, given subgraph

\[
\text{test: } k \in m \quad \text{return: } 2 \cdot m(k)
\]

of the dependence graph in Figure 12 a \( v = \text{get}(k) \) operation installed at node \( k \in m \) also covers the \( m(k) \) expression required by the successor return: \( 2 \cdot m(k) \) node. In the actual implementation, the local variable \( v \) that stores the value of expression \( m(k) \) flows into the multiplication operation, yielding the desired return value.

### 4.3 Complete Algorithm

The complete synthesis procedure is summarized in pseudocode form as Algorithm 1. We first (i) initialize an empty mapping \( \pi \) that we later use to record associations between concrete operations and the abstract operations they cover, and (ii) compute a dependence graph \( DG \) over the behavior graph \( G \). We then iterate over the nodes of \( G \) that represent abstract operations (rather than local expressions), as enforced by the \( \text{Abs} \) filter. For each node \( n \), we map \( n \) to the operations that are capable of generating \( n \)'s respective expression. For each operation \( c \), we compute a maximal region around \( n \) that \( c \) covers based on \( DG \). This yields a mapping \( c \mapsto \{n, n_1, \ldots\} \) between \( c \) and the set of abstract operations it covers, which we record into \( \pi \).

Having iterated over all the nodes \( n \in \text{Abs}(N) \) and populated \( \pi \), we submit the abstract operations \( \text{Abs}(N) \) as well as the sets of operations maintained as the image of \( \pi \) (\( \text{Im}(\pi) \)) to a solver. This results in a cover, which is then mapped back to its respective concrete operations. The concrete operations are then composed according to the behavior graph \( G \) and the dependence graph \( DG \) to form the new implementation of the subject method. Composition is achieved by applying standard code transformations — namely, code isolation and extraction — while preserving program dependencies. The complete description is given in an accompanying technical report.1

The intuition informing our synthesis algorithm — of reimplementing the composed operation using the least amount of shared operations — is characterized in the following: Given composed operation \( m \), suppose the transformation rewrites some \( \text{Map} \) operation \( o \) in \( m \) as a local expression \( \ell \), which infers the behavior of \( o \) from the return value of an adjacent \( \text{Map} \) operation \( o' \), which represents the \( \text{Map} \) state at \( o' \). The transformation guarantees atomicity over \( o \) and \( o' \), because they operate on the same \( \text{Map} \) state as if the behavior of \( o \) takes effect instantaneously after \( o' \). Based on the amount of shared operations, we organize the version \( m \) and the transformed version \( m' \) into a partial order,\(^\text{1}\)

---

1 https://sites.google.com/site/flint3141/tr
Lemma 1 follows the common fast path idiom, where the path A ; B is the fast path. It provides a sound template for optimizations that preserve linearizability. Following Lemma 1, we derive the fast path from the original code. We first identify the branch if (A) B that satisfies data independence and ensures linearizable execution. This is done by invoking the linearizability checker (Section 2.5). We then check A for side effects. Lastly, we combine if (A) B with the fix F according to Lemma 1. As an illustration, consider the example in Figure 2 Branch if (A) B corresponds to the code in lines 2-5. We find that A does not update the Map, and therefore we combine if (A) B with our linearizable but inefficient fix in Figure 5 leading to the linearizable and efficient fix in Figure 6.

5.2 Flint-R: Adding Rollback Capabilities

Beyond the performance optimization, we have designed another enhancement to improve the completeness of Flint. If the initial fix produced by Flint fails in the linearizability checking, e.g., the fix containing multiple invocations, then another transformation is attempted that makes use of lightweight rollback facilities. We dub this enhancement Flint-R.

Flint-R is inspired by the optimistic concurrency-control framework of Kung et al. [17] (in particular, Sections 3 and 4 therein), which assumes that a transaction divides into two phases illustrated in Figure 13: first read and then write. The write phase, before applying any writes, first verifies that none of the reads has become invalidated. The write phase is assumed to be atomic. We provide an overview in Appendix B.

Flint-R follows Kung et al.'s framework by classifying Map operations into two categories: read operations such as get, which do not update the Map state, and write operations such as put and putIfAbsent, which may update the Map state. Flint-R builds a transaction around each write operation by treating its execution as the write phase and the preceding execution as the read phase. In this way, Flint-R keeps the write phase short and minimizes the overhead incurred by the mutual exclusion. Overall, there may be multiple transactions for a composed method.

Importantly, Flint-R requires that the preceding execution has no global side effects, i.e., it should not write to any shared program states including the Map and other shared data structures. Intuitively, this requirement guarantees the execution may fail in the validation for multiple times without changing the program states until it finally passes the validation. We check the requirement through the path-sensitive data-flow analysis, which distinguishes the side effects of the conditional write operations under different path conditions. For example, as illustrated in pattern (1) of Figure 14, the conditional write operation putIfAbsent has no side effects if its return is not null according to the path condition. The above requirement excludes the case that one write op-

---

**ALGORITHM 1:** Pseudocode description of the Flint synthesis algorithm

Input: semantic behavior graph $G = (N, E)$  
Output: concrete method $m$ implementing $G$  

1. $\pi \leftarrow \{ \}$  
2. $DG = (N, E_{DG}) \leftarrow$ compute data dependencies over $G$  
3. foreach node $n \in \text{Abs}(N)$ do  
   4. $\{c_1, \ldots\} \leftarrow$ operations generating expression at $n$  
   5. foreach $c \in \{c_1, \ldots\}$ do  
      6. $\{n, n_1, \ldots\} \leftarrow$ neighbors covered by $c$ in $DG$  
      7. $\pi \leftarrow \pi \cup \{c \mapsto \{n, n_1, \ldots\}\}$  
   end  
9. end  
10. $S \subseteq \text{Im}(\pi) \leftarrow$ submit ($\text{Abs}(N), \text{Im}(\pi)$) to solver  
11. $\{c_1, \ldots, c_k\} \leftarrow$ map $S$ back to concrete operations  
12. $m \leftarrow$ implement $G$ using $\{c_1, \ldots, c_k\}$  
13. return $m$

$m \prec m'$.

Following the partial orders, the greater versions have more atomicity, and the maximal versions that contain only a single shared operation guarantees atomicity.

5. Enhancements

Beyond the core synthesis algorithm, Flint features facilities to boost the performance of the synthesized method, as well as handle certain situations where the synthesized implementation is still not linearizable. We describe both in the following.

5.1 Performance Optimization

While the fix in Figure 5 is correct, it may lead to considerable performance degradation. This is because $\text{computeV}()$ is always invoked, even if the respective key is already mapped under the Map.

To achieve a fix that has the same level of efficiency as the original code, we opportunistically attempt to optimize the core fix. Optimization transformations must, of course, preserve the atomicity of the subject operation. In the following lemma, we characterize a simple yet common pattern for optimization, which also guarantees the linearizability. The proof for this lemma is provided in Appendix C.

**Lemma 1.** Given a method in the form

$$
\text{if (A) \{ B \} else \{ C \}}
$$

and the linearizable fix $F$ produced by Flint,

$$
\text{if (A) \{ B \} else \{ F \}}
$$

is also linearizable if,

1. The execution following the path $A ; B$ satisfies the data independence criterion and is linearizable.
2. The execution of $A$ does not update the map.
operation follows another successful write operation in a path. Flint-R does not apply if the requirement is not satisfied.

Flint-R is further augmented with the transactional boosting according to Map semantics. That is, the validation checks the conflict among transactions at the level of the abstract memory locations (Section 4.2), rather than the level of concrete memory locations. This eliminates the spurious conflicts. For example, given the transaction with the write operation map.put(5,5) and the transaction with the read operation map.get(6), they may conflict at the concrete memory level because both access the internal field ConcurrentHashMap$HashEntry.hash. However, they do not conflict with each other at the abstract memory level as they involve independent mappings of different keys. From the implementation perspective, the above validation is implemented following the common practice of abstract locks.

6. Limitations

Limitations shared with verification Data independence (Section 2.5) is a clear limitation of both the verification work and our work, which restricts generality. Specifically, the checker can check the linearizability (correctness) only under the data independence conditions. In case that these conditions do not hold, the checker conservatively reports the composed operation as non-linearizable.

Limitations specific to our work As also specified in Section 7.1, our work has its own limitations: (1) it focuses on the composed operation involving a single shared Map object. We do not consider the composed operation involving additional shared collections because the linearizability would be related to all the collections and reasoning about a single Map object cannot guarantee the overall linearizability. (2) It does not consider the composed operations involving global operations such as size, clear and putAll, which access the global state of the Map object. Such composed operations require the global synchronization over the entire Map object, e.g., protection of the Map object using the locks or validation of the global Map state using the rollback facilities.

Besides, our two enhancements (Section 5) apply only in the specific conditions. The performance optimization, i.e., Lemma 1, applies only when (1) the fast path A ; B satisfies the data independence and is linearizable, (2) the condition checking in the fast path is free of side effects. Intuitively, the requirements specify that the execution may finish fast following the fast path, and if the condition for the fast path does not hold, the execution still proceeds normally starting with the unchanged program states. The requirements are used in the proof of Lemma 1 for ensuring the correctness.

The rollback recovery, i.e., Flint-R, applies only when the execution prior to the write phase has no global side effects. Intuitively, this requirement guarantees the execution may fail for multiple iterations without changing the program states until it is finally validated.

7. Evaluation

Our evaluation addresses three primary questions: (1) Does our approach have general applicability? (2) Is the fix efficient? (3) Does our fix outperform general solutions for composing operations, such as software transactional memory (STM)?

7.1 Experimental Design

Prototype Implementation We have implemented a prototype of the Flint and Flint-R algorithms. Flint is built atop the Soot framework and extends the available-expression analysis provided by Soot. To solve the cover problem, Flint invokes the Lpsolver tool. Since Lpsolver does not support boolean connectives, Flint first replaces occurrences of the disjunction connective in the input formula with the sum and comparison arithmetic operators. For linearizability checking, we have implemented our own version of the checker designed by Shacham et al., which is specialized for Maps.

Experimental Setting Our experiments and measurements were all conducted on an x86.64 Thinkpad W530 workstation with eight 2.30GHz Intel Core i7-3610QM processors, 16GB of RAM and 6M caches. The workstation runs version 12.04 of the Ubuntu Linux distribution, and has the Sun 64-Bit 1.7 Java virtual machine (JVM) installed.

All the performance test drivers we created, as well as the collected data, are publicly available. We link to public artifacts from relevant contexts. In our experiments, we repeated every configuration 21 times. The reported statistics reflect the last 20 runs, excluding the first (cold) run that performs general initialization logic.

Benchmarks Our study covers 48 real-world composed operations that suffer from linearizability violations. These are taken from a diversified set of 27 real-world applications, including Tomcat, Cassandra and MyFaces, as shown in Table 4 in Section A.

These operations are a subset of the full list analyzed by Shacham et al. after eliminating methods that are (i) linearizable, (ii) nonlinearizable but modify other shared collections beyond the shared Map object, or (iii)
contain global operations, such as size, clear and putAll, which involve the entire Map object. The methods in (ii) are excluded because our approach focuses on composed operations involving a single Map object. The methods in (iii) require global synchronization over the entire Map object. We have made the source code of the extracted operations publicly available.

### 7.2 Aggregate Statistics

We visualize statistics on successful fixing by both Flint and Flint-R as a pie chart. Flint is able to successfully fix 79% of the buggy methods (38/48), and the combination of Flint and Flint-R is able to handle 96% of the bugs (46/48) with only 2 bugs being beyond the reach of both Flint and Flint-R. Beyond the core Flint algorithm, Flint-R is able to produce a fix in 17% of the cases where Flint fails. In isolation, Flint-R produces fixes only 24% of the buggy methods (16 in common with Flint), which comes to 50% of all benchmarks.

A more detailed view of the results is provided in Table 3. Each entry in this table refers to a different cluster of buggy methods, whose characterization (explained in detail later, in relevant context) is given in the leftmost column and the bugs included in it are given in the middle column. The rightmost column indicates which of Flint and Flint-R were able to fix the problem (either one of the algorithms or both or none).

In all cases, the Flint algorithm completes in under 100 milliseconds. The main reason for this is the modular nature of Flint, which refrains from modeling other threads and thread interleavings. Breakdown of the overall Flint running time into steps reveals that data-flow analysis converges within 50 milliseconds, and solving the cover problem usually requires up to 20 milliseconds. Data-flow analysis converges fast, albeit being path sensitive, because the buggy methods mostly feature simple control flow (often without loops). The solver returns a solution quickly because there are typically only few abstract operations (≤ 10) and covers (≤ 5) per buggy method.

These compelling performance statistics for the Flint algorithm, combined with the high frequency and quality of successful fixes, suggest Flint as effective IDE-hosted refactoring tool. In this use case, the developer highlights a method, which is checked for linearizability, and — if the method is confirmed to violate linearizability — refactored by Flint.

![Pie Chart](https://sites.google.com/site/flint3141/)

### 7.3 Detailed Analysis

#### Bugs Fixed by Flint

Beyond the raw statistics, we manually examined the bugs, and classified those fixed by Flint into a catalog of access patterns. These reflect idiomatic composition that leads to linearizability violations. The resulting catalog, which consists of 12 categories, is provided in Figure 14. The respective fixes appear in Figure 15. For consistency, we normalize variable names: k is a key into the Map, v, v2, .. and o are values, and the Map reference is m.

For pattern (a), Flint is able to eliminate the get operation after putIfAbsent. The first get operation is removed due to minimization, but then reinserted as an optimization. (See Section 5.1). A similar fix is applied for pattern (b), which has a slightly more sophisticated behavior. The fix for pattern (c) again utilizes the putIfAbsent interface, though no get calls are dropped. The fix for pattern (d) drops both the containsKey and get. We cannot optimize pattern (d) further as no path in it ensures linearizable execution.

Patterns (e) and (f) have the same sequential behavior, and thus also the same fix. The get optimization in the buggy version is featured also in the fix by Flint, and the return value is determined based on the output by putIfAbsent. For pattern (e), the solver (described in Section 4.1) enforces the use of putIfAbsent rather than put. Pattern (g) is reminiscent of pattern (d), and thus corresponds to the same Flint fix. Pattern (h) contains a loop, though this loop always exits after the first iteration in a sequential setting. Hence, Flint eliminates the loop from the fix.

For pattern (i), Flint first eliminates the get call by relying solely on putIfAbsent. get is then restored as an optimization. The fix for pattern (j) drops the call to containsKey and instead uses only get. containsKey

<table>
<thead>
<tr>
<th>Characterization</th>
<th>Bug IDs</th>
<th>Fixed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern (a)</td>
<td>1,39,46</td>
<td>Flint</td>
</tr>
<tr>
<td>Pattern (b)</td>
<td>2,3</td>
<td>Flint</td>
</tr>
<tr>
<td>Pattern (c)</td>
<td>5,20,23,27, 32,34,37,40, 41,45,47,48</td>
<td>both</td>
</tr>
<tr>
<td>Pattern (d)</td>
<td>8,17,18,35</td>
<td>Flint</td>
</tr>
<tr>
<td>Pattern (e)</td>
<td>9,28,29, 38,42,43</td>
<td>Flint</td>
</tr>
<tr>
<td>Pattern (f)</td>
<td>24,25</td>
<td>Flint</td>
</tr>
<tr>
<td>Pattern (g)</td>
<td>22,26,30,31</td>
<td>Flint</td>
</tr>
<tr>
<td>Pattern (h)</td>
<td>33</td>
<td>both</td>
</tr>
<tr>
<td>Pattern (i)</td>
<td>4</td>
<td>both</td>
</tr>
<tr>
<td>Pattern (j)</td>
<td>14</td>
<td>both</td>
</tr>
<tr>
<td>Pattern (k)</td>
<td>10</td>
<td>both</td>
</tr>
<tr>
<td>Pattern (l)</td>
<td>11</td>
<td>Flint</td>
</tr>
<tr>
<td>Independent operations</td>
<td>6,7,36</td>
<td>Flint-R</td>
</tr>
<tr>
<td>Different keys</td>
<td>21,44</td>
<td>Flint-R</td>
</tr>
<tr>
<td>Intervening condition check</td>
<td>15,16,19</td>
<td>Flint-R</td>
</tr>
<tr>
<td>try/catch with side effects</td>
<td>12,13</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 3. Summary of successful bug fixes, where bugs are grouped according to their characterization.
\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{map-access-patterns.png}
\caption{Map access patterns arising in non-linearizable compositions}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{natural-fixes.png}
\caption{Natural fixes for non-linearizable compositions in Figure \ref{fig:map-access-patterns} (// opt denotes optimizations beyond the core fix.)}
\end{figure}
is later restored as an optimization. Similarly, in pattern (k) all operations except remove are dropped by Flint, but then containsKey is inserted as an optimization. Finally, in pattern (l) get is removed, and the result is instead based on the output of put.

**Bugs Fixed Only by Flint-R** There are 8 bugs that cannot be fixed using Flint, but can be fixed by Flint-R, as shown in Table 3. The coding idioms underlying these cases are listed in Figure 16. In pattern (1), the update operations putIfAbsent and replace have different semantics, and neither can be used to cover the other. An example of this pattern is bug 6, which cannot be rewritten by Flint. Flint-R applies because the preceding execution for either update operation is free of side effects according to the path-sensitive analysis. Note that the putIfAbsent operation does not write to the Map if the execution follows the if branch.

Pattern (2) refers to methods that access more than one key. Flint cannot use one concrete operation to cover different keys. An example of this is bug 21. Rollbacks provide a mechanism to deal with such situations.

Last, pattern (3) — manifesting e.g. in bug 15 — refers to the condition check c (line 3) intervening between the Map operations op1 and op2 (at lines 1 and 3). The dependencies op1 \rightarrow c and c \rightarrow op2 prevent one concrete operation from covering both, as they are even not adjacent in the dependence graph. Again, rollback facilities provide a solution.

An important constraint on the applicability of Flint-R is that the execution prior to the write phase should not carry any global side effects. For example, in pattern (1) in Figure 16 the execution until replace succeeds carries no side effects. This constraint on Flint-R is because its speculative behavior is limited to only rolling back local state, being less powerful (and thus also more lightweight) than STM systems.

The constraint also explains why Flint-R fails to fix many bugs, including most of those fixed by Flint, where intermediate evaluation steps produce side effects. As an example, pattern (g) — which Flint addresses by minimizing the number of used APIs — cannot be fixed by Flint-R because successful invocation of putIfAbsent creates side effects.

A final important point that we emphasize is that Flint-R is, in general, more applicable to the composition produced by Flint than to the original composition. That is, even if Flint proposes a candidate that is not simply linearizable, Flint-R can often recover linearizability for that candidate via speculation, but not for the original buggy method. The key reason for this is that abstracting the sequential behavior of the method, and then biasing synthesis toward minimality, has the pleasant side effect that there are less update operations and updates in general.

To illustrate this, we refer to Bug 36 in Figure 17. First, the else branch at lines 6-7 is removed by our abstraction because the branch is never taken in sequential executions. Second, abstraction merges the operations at lines 4 and 5 into a single replace invocation. This step is crucial to enabling Flint-R. Otherwise, bad interleavings between lines 4 and 5 cannot be recovered from due to the side effects caused by remove. Under abstraction, m.remove(k,v) becomes if(m(k)==null)m[k]↦null, and m.putIfAbsent(k,v) becomes if(m(k)==null)m[k]↦v. The overall behavior is then if(m(k)==null)m[k]↦v, which is covered by the replace API. The resulting fix is shown in Figure 18. Flint-R now applies because there is a single write operation and its preceding execution is side effect free.

**Bugs that are Beyond Flint and Flint-R** Bugs 12 and 13 cannot be fixed neither by Flint nor by Flint-R. This is because there are Map operations op1 and op2 that are placed in the try and catch blocks separately. Denote the catch block by c. Then the dependencies op1 \rightarrow c and c \rightarrow op2 prevent one operation from covering the other, as they are not even adjacent in the dependence graph. As for Flint-R, the code between the operations involves updates to shared state outside the Map such as logs, which cannot be rolled back.

### 7.4 Performance Measurements

We now turn to measure the performance characteristics of Flint fixes.

**Significance of Optimization** The first experiment measures the runtime overhead of fixes produced by Flint with and without optimization. We refer to the optimized version — which is available for most patterns but not for (d), (g) and (l) — as Flint-O. For this experiment, we created a test driver for every composed operation, in which multiple threads running concurrently share a fixed-size workload (varying slightly between operations to align running times). The workload consists of random inputs to the operation with the exception of patterns (j) and (k), for which the driver first populates the Map. (Otherwise certain functionality, like lines 3–4 in pattern (k), is never executed.)

The results are depicted in Figures 19a,19b. The graphs visualize running time as a function of the concurrency level, which peaks at 8 threads because this is the number of cores on the machine. In general, Flint-O fixes incur negligible overhead (less than 1%), and may even outperform the original version as certain invocations are dropped (like the get
call in pattern (f)). Flint fixes, on the other hand, may incur up to 50% overhead, highlighting the importance of optimization. For pattern (d), an optimized fix cannot be computed by our transformation algorithm, and indeed the fix is 44%-62% slower than the original code. Figure 20, which plots running time as a function of workload size for pattern (f), demonstrates that without optimization, slowdown may grow proportionally to the number of threads. In the case of pattern (f), this is because compute is invoked even if the key is in the Map.

Comparison with General Techniques The second experiment compares between Flint/Flint-O (where Flint-O is used when different from Flint), Flint-R and two other variants. The first is STM, which is a general synchronization approach that guarantees atomicity optimistically by recording shared memory accesses and rolling back bad executions. The second is the abstract lock synchronization at the granularity of Map keys [7,9], which is reminiscent of our dataflow analysis. For STM, we used the Deuce library [8] and in particular, the default Deuce protocol, called TL2 [6], which applies eager speculation. For this experiment, we used high-contention workloads to study the different approaches under high stress. This was achieved by (i) increasing the number of operations while narrowing the key pool, as well as (ii) focusing the test drivers on operations that do not commute with the operations in the fixed version according to the heuristics proposed by Shacham et al. [27].

The results are presented in Figure 21. For patterns sharing the same fix, we show the comparison for only one representative. As the graphs indicate, when there are more than 8 threads, our fix is generally (i) around 5% faster than the rollback fix (Figures 21a, 21d, and 21), (ii) 14%-20% faster than the locking fix (Figures 21a, 21b, and 21d), and (iii) 15%-26% faster than the STM fix (Figures 21a, 21b, 21d, and 21e). These differences in performance cannot be seen with ≤ 4 threads, which suggests that they are associated with the concurrency of the fix rather than its overhead (except STM, which incurs visible overhead at all concurrency levels due to memory instrumentation and runtime checks).

8. Related Work

Blocking Synchronization Both dynamic approaches [4,23] and static approaches [13,20,35] have been proposed to fix atomicity violations. Dynamic methods typically leverage architecture support for detection of bugs, which incurs observable runtime overhead. Prevention of atomicity violations is achieved at runtime by delaying the execution of some thread. Static approaches typically introduce new locks to fix atomicity violations, which limits concurrency and may also lead to deadlocks.

Nonblocking Synchronization Nonblocking approaches — most notably STM — have also been proposed for recovery from atomicity violations [21,36]. These require conflict detection and rollback facilities. Conflict detection at the field level is overly conservative for Map updates because irrelevant fields, such as hash or modCount, are written by operations accessing different keys [32]. This leads to unnecessary rollbacks. Zhang et al. [57] propose idempotence-based rollback, which does not require checkpointing, similarly to Flint-R. Their rollback, designed for general use, requires the programmer’s help in identifying the failure sites, while our approach automatically identifies failures through the standard Map semantics. Besides, we leverage Map semantics and path sensitivity to precisely reason about the side effects. As an example, putIfAbsent has no side effects when it returns a non-null value.

Map Semantics Synchronization at the level of abstract Map semantics has been utilized to improve STM [9,16,30,31]. This helps in reducing the false alarms due to spurious conflicts. Similarly to our approach, the approach proposed by Lin et al. [18] also leverages Map semantics to fix incorrect compositions. However, their approach is based on syntactic pattern matching, and is therefore limited to predefined templates. Besides, their approach guarantees the absence of runtime exceptions, but may lead to fixes that are functionally incorrect fixes (like the one in [4]).

Privatization Privatization transformations [12,24,33] have also been proposed to fix atomicity violations. Privatization turns shared reads to local reads to avoid buggy interleavings, similarly to the concept of Flint. Flint additionally features inference logic, with which it automatically resolves which Map accesses to privatize.

Synthesis Our approach can be used for synthesis, the sequential code serving as the specification. Vechev et al. [34] propose a semi-automatic approach to synthesize the linearizable implementation in general. Their approach, however, interactively involves the programmer. Our approach, instead, is built on the standard semantics of Maps, and with
Figure 19. Overhead

(a) Patterns (a), (b), (c), (e), (f) and (h) (Pattern c as the representative)

(b) Patterns (d) and (g) (Pattern d as the representative)

(c) Pattern (i)

(d) Pattern (j)

(e) Pattern (k)

(f) Pattern (l)

Figure 21. Performance comparison

(a) Patterns a, b, c, f and h (Pattern c as the representative)

(b) Patterns d and g (Pattern d as the representative)

(c) Pattern i

(d) Pattern j

(e) Pattern k

(f) Pattern l
this focus we do not rely on user interaction. Hawkins et al. \cite{Hawkins2010} automatically synthesize, based on a relational specification, a concurrent lock-based data structure. Our approach instead leverages safe compositions of concurrent Map APIs. Besides, at the conceptual level, our technique is similar to the instruction selection technique \cite{Appel2002} (Chapter 9), a classical compiler technique that finds a cover for the primitive instructions using the real machine instructions. Each real machine instruction typically performs several primitive instructions. However, our technique is different from the instruction selection technique: (i) we address Map operations rather than low-level instructions, and (ii) we solve the cover problem at the abstract rather than concrete level.

9. Conclusion and Future Work

We have designed and evaluated Flint, the first general algorithm for automatically transforming nonlinearizable compositions of Map operations into atomic compositions. Our approach performs semantic modeling of the composed operation, then reimplements it using a minimal set of Map APIs, and finally applies optimizations. Evaluation of Flint on 48 incorrect compositions from 27 popular applications yields strong evidence in favor of our approach. In the future, we intend to generalize Flint beyond Maps via a general specification format for concurrent data structures with ADT semantics. We would also like to extend Flint to support global operations (like `size`).

References

\begin{enumerate}
\item D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In DISC, 2006.
\item M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In OOPSLA '06.
\item Y. Lin and D. Dig. Check-then-act misuse of java concurrent collections. In ICST, pages 164–173, 2013.
\item O. Shacham. Data structures in the multicore age. CACM, 2011.
\end{enumerate}
A. Linearizability

Linearizability Formally, an operation $op$ consists of an invocation event $(t, op, i)$, where $t$ denotes the thread identifier, $op$ is the operation identifier, and $i$ are the arguments, and a response event $(t, op, r)$, where $r$ is the return value. Given operation $op$, we denote its invocation (resp. response) by $inv(op)$ (resp. $res(op)$). We consider finite sequences of invocation and response events, known as histories.

A complete invocation of operation $op$ is the two-event history $[inv(op), res(op)]$. A complete history is a history where every invocation event is matched by a response event. A sequential history is the composition of one or more complete invocations. A thread subhistory $h|_t$, is the projection of history $h$ on the events where $t$ is incident. We assume a sequential specification for the concurrent object, and thus the legality of a sequential history is decided by the specification. We refer to histories $h$ and $h'$ as equivalent if $\forall t. h|_t \equiv h'|_t$. Given operations $op$ and $op'$ in history $h$, we say that $op$ precedes $op'$ and write $op <_h op'$ if $res(op)$ appears before $inv(op')$ in $h$.

A history $h$ is linearizable if there exists a legal sequential history $h'$, such that

$$h \equiv h' \land \forall op, op'. op <_h op' \Rightarrow op <_{h'} op'$$

That is, $h'$ is equivalent to $h$, and further respects the global ordering of non-overlapping operations in $h$. Finally, a concurrent object is linearizable if all the histories over that object are linearizable.

B. Kung et al.’s Framework

The method developed by Kung et al. enforces serial execution of the write phases using the mutual exclusion synchronization, i.e., the lock. To achieve atomicity of the transactions, it further requires, for any two transactions $T_i$ and $T_j$, either (1) the transaction $T_i$ finishes the write phase before the transaction $T_j$ starts the read phase, or (2) $T_i$ finishes the write phase before $T_j$ starts the write phase and the write set of $T_i$ does not intersect the read set of $T_j$. If the write set and the read set intersect each other, we say the transactions $T_i$ and $T_j$ conflict.

From the implementation perspective, Kung et al. have designed a counter-based approach to assign unique transaction identifiers and identify the transactions that may conflict with the current transaction, i.e., the transactions that finish the write phase after the current transaction starts the read phase but before it starts the write phase. The method by Kung et al. then validates whether the identified transactions and the current transaction conflict. An important property of the framework is it does not need the heavyweight checkpointing to support rollbacks. If validation fails, there are no special actions to be taken except restarting the transaction, because the preceding history consisted only of read operations.
C. Proofs

C.1 Proof for Lemma 1

Proof. We begin by giving an informal intuition. Intuitively, the path $A;B$ acts as a shortcut which allows the execution to return fast. In case that the execution does not follow the shortcut path and takes the other branch, the code $F$ resumes the execution, being oblivious to the state information acquired at $A$, as if $A$ is not executed. Executing $F$ alone suffices to ensure the correctness. Specifically, suppose the thread follows the path $A;F$ but is interleaved by another operation $\ast$, i.e., $A;\ast;F$. Because the test performed in $A$ is also done separately within $F$, the behavior of $F$ is independent of the evaluation of $A$. Hence, $A;\ast;F$ is equivalent to $\ast;F$, which is the sequential execution of the fix $F$.

For the full proof, we need to fix the formal setting. We have a main thread executing the composed operation $C$ and other (environment) threads executing atomic operations. Let $<l,m>$ denote the program state, where $l$ denotes the state of the local variables (including program counter pc) in $C$ and $m$ denotes the map state. Three types of state transitions exist: (1) local transition (Equation 3) and (2) main transition (Equation 4) model the basic execution steps of the main thread, which correspond to local operation and Map operation, respectively; (3) environment transition (Equation 5) models a Map operation, or specifically Map update [(27)], executed by an environment thread.

$$<l,m> \xrightarrow{\ast} <l',m>$$ (3)

$$<l,m_i> \xrightarrow{x=op(args)} <l[x \rightarrow \text{ret}],m_{i+1}>$$ (4)

$$<l,m_i> \xrightarrow{m.op(args)} <l,m_{i+1}>$$ (5)

The execution may follow two paths, $A;B$; or $A;F$; where the former is required to be linearizable and the latter is to be proven as linearizable. Consider the execution $A;\ast;F$, where the interleaving operation $\ast$ happens just after the main thread enters the branch that contains $F$. Equation 6 shows the state transitions.

$$<l_0,m_0> \xrightarrow{A} <l_1,m_0> \xrightarrow{\ast} <l_1,m_1> \xrightarrow{F} <l_2,m_2>$$ (6)

$$<l_0,m_0> \xrightarrow{\ast} <l_0,m_1> \xrightarrow{A} <l_1,m_1> \xrightarrow{F} <l_2,m_2>$$ (7)

$$<l_0,m_0> \xrightarrow{\ast} <l_0,m_1> \xrightarrow{A} <l_2,m_1> \xrightarrow{B} <l_3,m_2>$$ (8)

The interleaved execution is equivalent to the sequential execution $\ast;A;F$ or $\ast;A;B$. Which branch to take, $B$ or $F$, depends on the state after $\ast$. If the sequential execution is $\ast;A;F$, the state transitions, shown in Equation 7 are equivalent to transitions in Equation 6 because $F$, as an independent fix, does not read any locals defined by $A$, and the input states to $F$ in Equation 6 and Equation 7 are equivalent.

If the sequential execution is $\ast;A;B$, the state transitions are shown in Equation 8. The input state to $A;B$ in Equation 8 and the input to $F$ in Equation 6 are identical except they have different local states. Recall that $F$ does not read the locals defined by $A$ (Equation 6), therefore, $F$ and $A;B$; essentially start with the same local states, i.e., the empty states. Besides, the executions following both $F$ and $A;B$; are linearizable, transiting to the same output state as the sequential execution does.

$\square$

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptive planning</td>
<td>Tool for corporate budgeting</td>
</tr>
<tr>
<td>Adobe BlazeDS</td>
<td>Server-based remoting tool</td>
</tr>
<tr>
<td>Annsor</td>
<td>Runtime annotation processing tool</td>
</tr>
<tr>
<td>Apache Cassandra</td>
<td>Distributed NoSQL database</td>
</tr>
<tr>
<td>Apache MyFaces Trinidad</td>
<td>JSF framework</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>Java Servlet container</td>
</tr>
<tr>
<td>ApacheCXF</td>
<td>Services framework</td>
</tr>
<tr>
<td>Carbonado</td>
<td>Persistence abstraction layer</td>
</tr>
<tr>
<td>CBB</td>
<td>Concurrent building blocks</td>
</tr>
<tr>
<td>dyuproject</td>
<td>REST framework</td>
</tr>
<tr>
<td>Ektorp</td>
<td>Persistence layer on top of CouchDB</td>
</tr>
<tr>
<td>EntityFS</td>
<td>File system library</td>
</tr>
<tr>
<td>fleXive</td>
<td>Content repository</td>
</tr>
<tr>
<td>GlassFish</td>
<td>JavaServer faces</td>
</tr>
<tr>
<td>gridkit</td>
<td>Data grid library</td>
</tr>
<tr>
<td>GWTEventService</td>
<td>Event-based communication framework</td>
</tr>
<tr>
<td>Hazelcast</td>
<td>In-memory data grid</td>
</tr>
<tr>
<td>Hudson</td>
<td>Continuous integration (CI) tool</td>
</tr>
<tr>
<td>JRipples</td>
<td>Change impact analysis tool</td>
</tr>
<tr>
<td>memcache-client</td>
<td>Java memcache client</td>
</tr>
<tr>
<td>P-GRADE</td>
<td>P-GRADE Grid Portal</td>
</tr>
<tr>
<td>Tammi</td>
<td>JMX-based development framework</td>
</tr>
<tr>
<td>RESTEasy</td>
<td>REST framework</td>
</tr>
<tr>
<td>Retrotranslator</td>
<td>Java compatibility tool</td>
</tr>
<tr>
<td>Torque-spring</td>
<td>Torque support classes</td>
</tr>
<tr>
<td>Vo Urp</td>
<td>UML Representation Pipeline</td>
</tr>
<tr>
<td>WebMill</td>
<td>CMS portal</td>
</tr>
</tbody>
</table>