View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Staccato: A Bug Finder for Dynamic Configuration Updates*

John Toman[1] and Dan Grossman[2]

1 Department of Computer Science & Engineering
University of Washington
Seattle, WA, USA
jtoman@cs.washington.edu

2 Department of Computer Science & Engineering
University of Washington
Seattle, WA, USA
djg@cs.washington.edu

Abstract

Many modern software applications are highly configurable, allowing configuration options to be changed even during program execution. When dynamic configuration updating is implemented incorrectly, program errors can result. These program errors occur primarily when stale data—computed from old configurations—or inconsistent data—computed from different configurations—are used. We introduce Staccato, the first tool designed to detect these errors. Staccato uses a dynamic analysis in the style of taint analysis to find the use of stale or inconsistent configuration data in Java programs. It supports concurrent programs on commodity JVMs. In some cases, Staccato can provide automatic bug *avoidance* and semi-automatic repair when errors occur.

We evaluated Staccato on three open-source applications that support complex reconfigurability. Staccato found multiple errors in all of them. Staccato requires only modest annotation overhead and has moderate performance overhead.

1998 ACM Subject Classification D.2.9; Software configuration management

Keywords and phrases Dynamic Configuration Updates, Dynamic Analysis, Software configuration

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.24

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
http://dx.doi.org/10.4230/DARTS.2.1.14

1 Introduction

Today's software is highly configurable [42, 43, 17]. This configurability increases the reusability and portability of software. Software configurations can be specified in several different ways: static compile-time configuration options [20], command-line options that can be changed on each program run [37], or options specified in a configuration file read at startup. Along with high levels of configurability, software is increasingly subject to stringent uptime requirements; restarting an application to effect a configuration change is not

* This paper is based upon work supported in part by the National Science Foundation under Grant No. CCF-1064497AM001, and upon research sponsored in part by DARPA under agreement number FA8750-16-2-0032.

© John Toman and Dan Grossman;
licensed under Creative Commons License CC-BY

always feasible. To accommodate configurability and robustness, many applications support configuration options that can be changed at runtime. We call a configuration change at runtime a *dynamic configuration update* (DCU).

Unfortunately, it is difficult to implement dynamic configuration updates correctly. A bug[1] in Solr,[2] an open-source text search and indexing server, demonstrates this. The user may specify *analyzers* that process text before it is indexed. These analyzers apply case normalization, remove stop words, and so on. The user specifies analyzers in a configuration file read by Solr at startup. Solr also provides a reload command which re-reads the configuration file and re-initializes the system. However, the analyzers were not updated to reflect the new configuration, even though Solr reported the reload complete. Solr would then silently misprocess data and incorrectly answer user queries. The only indication that something was wrong was Solr's output, which required careful inspection on the part of the user. We found similar defects in multiple applications.

This paper presents, to the best of our knowledge, the first study of defects in dynamic configuration update implementations. We have developed a dynamic analysis that can assist developers in finding and diagnosing defects in their DCU implementations. We implemented our technique in Staccato (STAle Configuration and Consistency Analysis Tool), a prototype DCU error detection tool for Java programs.

Our approach checks one of two alternative correctness conditions for DCU systems, as chosen by the programmer. The first condition states that old versions of configuration options may not be used after a configuration update. The second states that only one version of a configuration option may be used during a single method execution. Both conditions provide a possible specification of program behavior in the presence of dynamic configuration updates. Choosing the appropriate correctness condition for a program unit requires domain knowledge. We have the user of Staccato select the appropriate condition with a few lines of high-level annotations (fewer than 0.7 per 1,000 SLOC in our evaluation).

Underlying both conditions is the notion of *versioning* the software configuration. For every program value, Staccato tracks which configuration options, and what versions of those options, were used to construct that value. This information moves with a value as it transformed and combined with other values. Whenever a value is read by the program, Staccato checks that the value does not violate the correctness condition selected by the programmer. Our analysis supports concurrency and requires no changes to the JVM.

In addition to bug finding, Staccato provides support for program repair and bug avoidance. Staccato can transparently repair program schedules to hide insufficient synchronization around configuration accesses. In addition, when Staccato detects a value built from an out-of-date configuration, it calls a programmer-provided callback to update the stale value. We used this functionality to repair DCU errors that we discovered in our evaluation. We also found this technique to be effective enough that for some projects, we were able to add DCU support for options that previously required a restart.

We applied Staccato to three large open-source projects with extensive support for dynamic configurability. These were code-bases with which we were not previously familiar, but we were still able to effectively use Staccato to find DCU defects in all of them.

In summary, this paper makes the following contributions:

- We provide two correctness conditions for software that supports dynamic configuration updates (DCU).

[1] https://issues.apache.org/jira/browse/SOLR-3587
[2] http://lucene.apache.org/solr/

```
1 class RequestManager implements Reloadable {
2   String targetIp, apiKey;
3   RequestManager() {
4      targetIp = Config.get("request.target-ip");
5      apiKey = Config.get("request.api-key");
6   }
7   String doRequest() {
8      Request req = new ApiRequest(targetIp);
9      req.send(apiKey);
10     return req.response();
11  }
12  void reloadConfig() {
13     targetIp = Config.get("request.target-ip");
14  }
15 }
```

Figure 1 A bug caused by an incomplete configuration update: after an update `reloadConfig()` does not read the updated `"request.api-key"` option.

```
1 class SpotService implements WebService {
2   String handleSpotRequest() {
3     return Config.get("verb") + " Spot, " + Config.get("verb") + "!";
4   }
5
6   void handleUpdateRequest(String newVerb) {
7     Config.set("verb", newVerb);
8   }
9 }
```

Figure 2 A bug caused by an inconsistent view of the configuration: the `handleSpotRequest` method can observe two versions of the `"verb"` option in one execution.

- We define the problem of detecting and diagnosing errors in DCU code.
- We describe how errors in DCU implementations can be discovered with a novel dynamic information-flow tracking approach.
- We describe STACCATO, a tool implementing this approach for Java programs.[3]
- Using STACCATO, we show that bugs in DCU implementations affect multiple open-source projects, and that our technique is effective for finding these errors.

The rest of the paper is organized as follows. Section 2 introduces and justifies the correctness conditions checked by our approach. Section 3 introduces an information-flow tracking technique for detecting correctness violations in programs that support DCU. Section 4 details our approach for bug avoidance and program repair. Section 5 describes our implementation. Section 6 presents the results from applying STACCATO to three open-source software projects. Section 7 covers related work. Section 8 concludes.

2 Correctness Conditions for Dynamic Configuration Updates

STACCATO identifies errors caused by defects in dynamic configuration update mechanisms. We are unaware of any agreed upon correctness definition for DCU implementations either

[3] STACCATO is open-source. Our implementation and the tests we ran for this paper can be found at https://github.com/uwplse/staccato

in industry or the research literature (see Section 7). Accordingly, we first give two correctness definitions for DCU schemes that we developed after surveying configurable software systems. Both conditions provide a specification for program behavior in the presence of DCU. The programmer selects which condition to use for an application, with the option of switching conditions (or opting out altogether) on a per-method or per-class basis with annotations.

We begin with two motivating examples found in Figures 1 and 2. Both examples are invented code fragments that are representative of two common errors we found during our survey of configurable software. The `Config` class provides a mapping of configuration options to string values. In Figure 1, the user may update either the target IP or API key configuration options, after which the `reloadConfig()` method is called. The implementation has an error in its handling of the update: the `reloadConfig()` method fails to read the new value of the `"request.api-key"` configuration option. As a result, the configuration update requested by the user is not completely applied. The Solr bug described in the introduction is similar in form to this example.

Figure 2 shows a consistency issue that we frequently found in configurable software. In this example, `SpotService` implements a web service that can be configured with an administrative command. The primary functionality, expressed in the `handleSpotRequest` method, is to simply echo back sentences of the form, `"Run Spot, Run!"`. The verb in the response is controlled with the `"verb"` configuration option. An administrator can update the `"verb"` option by sending an update request that is handled by the `handleUpdateRequest` method. If an update request is received concurrently with a regular client request, a user can receive unexpected responses such as `"Run Spot, Bark!"` This error occurs because the use of `"verb"` in `handleSpotRequest` is not *atomic*: a single invocation of `handleSpotRequest` can observe two inconsistent versions of the `"verb"` configuration option.

Guided by these two examples and many others like them found in real programs, we have developed two conditions for software that supports DCU. The first condition is as follows:

▶ Correctness Condition 1 (Staleness). An execution must observe only values derived from the most *up-to-date* version of the program configuration.

We call correctness condition C1 the *staleness* condition as it states that a program may not use values built from stale configurations. This correctness condition is potentially violated in both examples. In the first example, after a configuration update the program will read a stale version of the `"request.api-key"` option on line 9 in Figure 1. In the second example, the string returned by `handleSpotRequest` may be derived (partially or completely) from an old version of the `"verb"` option.

While condition C1 is sufficient to identify the bug in Figure 2, it is overly strict. For example, consider again the scenario where the service in Figure 2 receives two concurrent requests handled by `handleSpotRequest` and `handleUpdateRequest` respectively. Due to nondeterminism in thread scheduling, `handleSpotRequest` may return the response `"Run Spot, Run!"` after `handleUpdateRequest` has updated `"verb"` to another value, such as `"Bark"`. This outcome violates condition C1, but some systems depend on allowing this behavior for high performance. A more desirable correctness condition would rule out only *inconsistent* outcomes, such as `"Run Spot, Bark!"`, which violate the programmer's expectations about configuration behavior. This observation motivates a second correctness condition:

▶ Correctness Condition 2 (Consistency). A method execution may observe only a *single* version of each configuration option.

```
1 String foo(String a, String b, boolean t) {
2   if(t) { return a; } else { return b; }
3 }
4 // V["foo"] = 2
5 String s = Config.get("foo"); // s^H = {"foo" → 2}
6 Config.set("foo", "..."); // V["foo"] = 3
7 String r = Config.get("foo"); // r^H = {"foo" → 3}
8 String q = foo(r, s, false); // q^H = {"foo" → 2}
9 q = foo(r, s, true); // q^H = {"foo" → 3}
```

Figure 3 Example configuration histories in a program. The history of the configuration value returned by `Config.get` on line 5 maps `"foo"` to the current version of `"foo"`, $\mathcal{V}[\texttt{"foo"}] = 2$. After the update on line 6, the epoch of `"foo"` is incremented and the value returned on line 7 is tagged with the new version. Notice that on lines 8 and 9 the configuration histories of `r` and `p` have moved through method parameters and return statements automatically.

In other words, a program may use an old version of a configuration option provided that version is not mixed with any other versions of the same option.

Both correctness conditions are useful in different situations. Which condition is appropriate for a given program is application dependent. In practice, we found that condition C2 (consistency) was a reasonable default for the programs in our evaluation set. Nevertheless, the staleness condition is a good fit for global, configurable objects, such as caches, database connections, etc. This led to a useful rule of thumb for using our technique: use the staleness condition for methods that manipulate configurable objects stored in static class members and use the consistency condition everywhere else. This was sufficient to find several bugs, and required fewer than 1 annotation per 1,000 SLOC across our evaluation set.

3 Technique

Staccato detects errors in DCU schemes with a dynamic information-flow analysis in the style of taint analysis. There are three pieces of Staccato's analysis, described in the next three subsections. First, Staccato versions the software configuration and associates each program value with a *configuration history* (Section 3.1). A value's history records which options, and what *versions* of those option, were used to build that value. Second, when values are combined, their configuration histories are merged to produce a history for the output value (Section 3.2). This *propagation* models the flow of configuration information through a program. Finally, whenever a value is read by the program, Staccato checks for violations of the DCU correctness conditions (Section 3.3). To ease explication, we will first describe our technique in terms of only detecting violations of the staleness condition (C1). Our approach for the consistency condition (C2) is presented in Section 3.4 as an extension.

3.1 Configurations Values

Staccato models the configuration of a program as a global map from strings to strings. Following standard terminology, we call each key an *option*. We call each value in the map a configuration *value*. Staccato also internally associates each configuration option with an *epoch* counter. These option epochs *version* the configuration values of a program. The epoch counters are stored in a map $\mathcal{V}$ from options to epochs: the current epoch of an option is denoted $\mathcal{V}[o]$. When an option o is updated, the value of $\mathcal{V}[o]$ is incremented by one. We will use e to denote arbitrary epoch values.

```
1 // V["foo"] = 2 and V["bar"] = 4
2 String f = Config.get("foo"); // f^H = {"foo" → 2}
3 String g = Config.get("bar"); // g^H = {"bar" → 4}
4 String h = f + g; // h^H = {"foo" → 2, "bar" → 4}
5 Config.set("foo", "..."); // V["foo"] = 3
6 String j = h + Config.get("foo"); // j^H = {"foo" → 2, "bar" → 4}
```

Figure 4 An example of history propagation and the history merge operation. On line 4, the input configuration histories have disjoint domains and therefore the merge operation is trivial. On line 6 the input configuration histories both have `"foo"` in their domains. According to the merge rule, the smaller of the two epochs for `"foo"` is chosen, which in this case is 2.

Staccato also tags each program value with a *configuration history*. The configuration history for a value v records the configuration options used to construct that value, along with the epochs (i.e., versions) of those options. The notation $v^{\mathcal{H}} = \{o_1 \to e_1, o_2 \to e_2, \ldots\}$ denotes that v was constructed using the value of configuration options $o_1, o_2, \ldots$ at versions $e_1, e_2, \ldots$ We will use $v^{\mathcal{H}}$ as a function mapping strings (option names) to epochs. $dom(v^{\mathcal{H}})$ denotes the set of options that appear in the configuration set $v^{\mathcal{H}}$. The notation $x^{\mathcal{H}}$ may be used when x is a variable to denote the history of the value referenced by x.

Configuration histories are associated with program values (not textual program variables). Thus, as a value moves through field array, variable assignments, or method boundaries, its configuration history automatically moves with it. This is important as DCU bugs involve complex information-flow: the Solr bug from the introduction involved 7 classes.

Configuration histories are empty for most values: only values that depend on the program configuration may have non-empty histories. Initially, the only values associated with non-empty configuration histories are configuration values themselves. When an option o is retrieved from the program's configuration, the returned configuration value is tagged with the history $\{o \to \mathcal{V}[o]\}$ (recall that $\mathcal{V}[o]$ is the current epoch of o). An example of this tagging and epoch counter updating can be found in Figure 3.

3.2 Propagation

Staccato tracks configuration values as they are combined with other values or transformed by the program. Thus, when multiple values are combined to create a new value, Staccato ensures that the new value's configuration history soundly captures all configuration information associated with the source values. The process of transferring configuration histories from operands to outputs is called *propagation*.

During propagation, configuration histories are merged together pairwise to yield the final configuration history of the output value. The merge operation for configuration histories is denoted $v^{\mathcal{H}} \cup u^{\mathcal{H}}$. If $dom(v^{\mathcal{H}}) \cap dom(u^{\mathcal{H}}) = \emptyset$, then the merge operation is a simple union. However, it is possible that some option o appears in both $v^{\mathcal{H}}$ and $u^{\mathcal{H}}$. Staccato conservatively handles this situation. Suppose that $v^{\mathcal{H}}[o] = e$ and $u^{\mathcal{H}}[o] = e'$. The output configuration history maps o to the epoch $min(e, e')$. This definition ensures that the epoch recorded for a configuration option o in a configuration history $x^{\mathcal{H}}$ is the oldest version of o used (either transitively or directly) to construct x. Figure 4 demonstrates history propagation and an example of the merge operation.

Staccato does not check for violations of the correctness condition during propagation. Recall that Staccato allows the programmer to select one of two possible correctness conditions or to opt out of checking altogether. If we integrated checking into propagation, we would have to implement three different propagation operations: one for each correctness

```
1 class DBConnection {
2   @StaccatoPropagate(RETURN)
3   static DBConnection connect(String host, String user) { ... }
4   String fetchRow(String query) { ... }
5 }
6 User getUser(...) {
7   String host = Config.get("db-host"), user = Config.get("db-user");
8   DBConnection conn = DBConnection.connect(host, user);
9   String userInfo = conn.fetchRow(...);
10  return new User(userInfo);
11 }
```

Figure 5 Propagation involving objects dependencies. The `@StaccatoPropagate` annotation the `connect` method propagates the configuration histories of the `host` and `user` parameters to the `DBConnection` object returned from the method. As a result, the configuration history of the `conn` in the `getUser` method has the `"db-host"` and `"db-user"` options in its configuration history.

condition and one for propagation without any checking. By separating the check and propagation operations, we can apply a uniform propagation operation across the entire program and vary the check operation depending on the correctness condition selected by the programmer.

Propagation for Object Types

A key design decision we faced is where propagation should occur. Operations such as integer arithmetic, boolean operations, and string concatenation all naturally propagate configuration information from source operands to output values automatically. Although sufficient for tracking simple dependencies, propagation involving only primitive values will miss dependencies at the object level. Consider the example in Figure 5. In order to verify that the usage of the connection object created on line 8 is correct, `conn` must inherit the configuration information of the `host` and `user` configuration values. Therefore, STACCATO also supports propagation from method arguments to outputs. This method-level propagation expresses configuration dependencies at a higher level of abstraction than what is possible with just primitive operations.

The user opts-in to method-level propagation on a per-method basis with annotations like the one seen on line 2 in Figure 5. The annotation argument indicates which output of the method is the target of propagation. There are two possible outputs: the return value (specified with the `RETURN` argument seen in the example) or the method receiver. The receiver output is used primarily for modeling side-effecting methods such as setters or constructors. After an annotated method is executed, STACCATO combines the configuration histories of the method's arguments with the existing history of the receiver or return value as appropriate. For the purpose of propagation to the return value, the method receiver (if applicable) is treated as an argument. In the database example, the propagation annotation ensures that $\texttt{conn}^{\mathcal{H}}$ contains the `"db-user"` and `"db-host"` options.

We experimented with an eager approach that propagated configuration histories unconditionally from method arguments to method receiver/return values. This seemed reasonable, as function outputs generally depend on input values. This strategy *would* properly capture the relationship between the `"db-host"` and `"db-user"` options and the `conn` object without user annotation. In practice, this approach suffers from a significant loss of precision. In Figure 5, automatic propagation also “taints” the value read from the database on line 9 with configuration information. By extension, the `User` object created on line 10 would also contain the database configuration options in its configuration history. However,

using a `User` object built with data read from an old connection is arguably *not* an error. This scenario is the expected behavior envisioned when a configurable database connection was implemented. In general, automatic propagation tags program data with unintuitive configuration histories, leading to false positives.

In our experience, configuration propagation is fairly rare; compared to the number of method definitions in our evaluation code-bases, the number of propagation annotations required was quite low, requiring fewer than 30 annotations across the entire evaluation set. We also found that propagation annotations need only be applied at natural API method boundaries, requiring no reasoning about whole program flow or API usage sites.

3.3 Checking

The final component of Staccato checks that the program does not observe values constructed from an out-of-date version of the configuration. Our approach actually checks an equivalent condition: that the value read by the program reflects only the most recent version of the configuration. A value v is up-to-date with respect to the current version of the configuration (as recorded in $\mathcal{V}$) iff: $\forall o \in dom(v^{\mathcal{H}}), v^{\mathcal{H}}[o] = \mathcal{V}[o]$. In other words, only the most recent configuration values may have been used (transitively or directly) to construct v. This check is performed on value reads. For the purposes of our analysis a value is read when it is: 1) passed as an argument to a method, 2) read from a field, 3) returned from a method, 4) read from a local variable, or 5) read from an array. If Staccato detects that the condition has been violated, we have identified a DCU defect and alert the user.

3.4 Extensions for Checking Consistency

We now discuss how to extend our approach to check for violations of the consistency condition (C2). Recall that the second correctness condition states that a method execution may not observe inconsistent versions of the program configuration. Checking this condition requires extending the representation of configuration histories and the merge operator.

In addition to option epochs, Staccato also tracks an extra bit for each option o in a configuration history. This bit is called the consistency flag. We denote this epoch-bit pair with $\langle e, f \rangle$, where e is an epoch and f is the new consistency flag. The consistency flag becomes set when Staccato detects that different versions of the same option have been used to construct a value. Any values derived (in terms of the propagation described in Section 3.2) from an inconsistent value are themselves marked as inconsistent. The consistency flag is initially unset for configuration values returned from the program's configuration: by definition, a configuration value always represents a single version of a configuration option.

The definition of the consistency flag gives rise to the following extension to the merge operator. Given two configuration histories to be merged, $u^{\mathcal{H}}$ and $v^{\mathcal{H}}$, such that $u^{\mathcal{H}}[o] = \langle e, f \rangle$ and $v^{\mathcal{H}}[o] = \langle e', f' \rangle$, the consistency flag in the merged configuration history is set iff: $e \neq e' \vee f = 1 \vee f' = 1$. The epoch for o is computed using the *min* function as previously described.

Given these extensions to configuration histories and the merge operation, checking the consistency condition is straightforward. The consistency condition requires that at most one version of each option may be used to build a value. This is precisely what the consistency flag tracks. Thus, checking a value v against the consistency condition entails checking that all consistency flags in $v^{\mathcal{H}}$ are unset. That is, a value v reflects a consistent view of the configuration iff: $\forall o \in dom(v^{\mathcal{H}}).v^{\mathcal{H}}[o] = \langle e, f \rangle \rightarrow f = 0$. This condition is checked at the same program points as the staleness condition.

```
1 int output = getValue();
2 while(...) {
3   if(Config.get("op") == "+") {
4     output += 2
5   } else {
6     output *= 2;
7   }
8 }
```

Figure 6 A simplified example of DCU error involving control-flow. If the `"op"` option is changed during the execution of the loop, the `output` variable will be processed inconsistently.

Control-Flow

We have so far discussed the consistency condition in terms of *values* produced and read by a program. This misses inconsistent behavior introduced by control-flow. A simplified example, based on a real bug found in our evaluation set, is shown in Figure 6. In this example, if the `"op"` property is updated before the loop has terminated, the `output` variable can reflect two different versions of the configuration. Lillack et al. [20] have noted that configuration values are often used in branching decisions so detecting errors involving control-flow is especially important.

To find errors like the one in Figure 6, we verify that the entire execution of a method observes a consistent view of the configuration. To check this property, upon entrance to a method we allocate a special sentinel object Σ. This object has a configuration history, which is empty at method entry. Unlike regular configuration histories, which record which configuration options and versions were used to construct a *single* value, $\Sigma^{\mathcal{H}}$ records the options and versions used by the *entire* method. Whenever a value v is read, $\Sigma^{\mathcal{H}}$ is updated such that $\Sigma^{\mathcal{H}'} = \Sigma^{\mathcal{H}} \cup v^{\mathcal{H}}$. $\Sigma^{\mathcal{H}'}$ is immediately checked for consistency using the rule described above. At method exit, Σ and its configuration history are discarded; each invocation begins with a fresh history. Method histories are not inherited by callers or callees.

To see how this approach catches errors involving control-flow, consider again the example in Figure 6. Assume at the entrance to the loop $\Sigma^{\mathcal{H}}$ is initially empty. On the first iteration of the loop, the `Config.get("op")` call on line 3 returns a configuration value with the configuration history $\{\texttt{"op"} \to \langle 1, 0\rangle\}$. This is merged with Σ's empty configuration history, giving $\Sigma^{\mathcal{H}} = \{\texttt{"op"} \to \langle 1, 0\rangle\}$. At the end of the first iteration, `"op"` is updated. On the second iteration, the value returned by the `get()` call on line 3 is a newer version of the `"op"` option, which is tagged with the history $\{\texttt{"op"} \to \langle 2, 0\rangle\}$. This history is merged with $\Sigma^{\mathcal{H}}$. The result of this merge is $\Sigma^{\mathcal{H}} = \{\texttt{"op"} \to \langle 1, 1\rangle\}$. Notice that the consistency flag for `"op"` is now set: this signals that the execution has now observed two conflicting versions of the `"op"` option. Staccato immediately reports this as an error.

The assumption that a method execution is the single unit for consistency is a heuristic, which may admit false positives and false negatives. A more precise approach would require precise interprocedural tracking of control-flow dependencies. We experimented with a version of Staccato that integrated an existing off-the-shelf implementation of precise control-flow tracking found in recent versions of Phosphor [4]. Although this approach was promising on small test examples, it suffered scalability issues when applied to the programs in our evaluation set. For example, after applying Staccato with control-flow tracking to a web application, the program failed to return a response to the client after 5 minutes. This overhead is due to the large amount of state maintenance required for precisely tracking control-flow dependencies. In contrast, our approach is relatively cheap to compute and can

find inconsistencies introduced through control-flow. In our experiments, this approach was not the source of any false positives.

4 Automated Bug Avoidance and Repair

Staccato also uses the techniques developed for checking to (semi-)automatically avoid DCU errors. Staccato supports two forms of program repair and automatic bug avoidance. The first mechanism (Section 4.1) automatically repairs buggy program *schedules* that would expose insufficient synchronization between configuration read and write operations. Staccato detects uses of configuration-derived values and automatically delays any configuration updates that would cause those values to become stale mid-use. Staccato also has a limited form of *value* repair (Section 4.2). The programmer may provide a function called by Staccato that repairs stale values when they are discovered. Both mechanisms target reducing violations of the staleness condition. In our experience, this condition is harder to get "right", and therefore benefits the most from automated assistance.

4.1 Coordinating Configuration Reads and Writes

In a multithreaded environment, it is possible for configuration updates to interfere with an ongoing use of a configuration-derived value. For example, a method may read some initially up-to-date variable x and then later re-read x. If an option in x's configuration history is updated between the two reads, Staccato would identify the second read as an error. It is assumed that the program has failed to appropriately react to the configuration update. However, the *use* of x—and by extension the options in x's configuration history—intuitively spans the two read operations. The error occurs because the *update* operation failed to wait for the outstanding use of x (and its corresponding configuration options) to finish. In general, an update of a configuration option being used in another thread without proper synchronization can lead to a DCU error.

Staccato can automatically *prevent* these errors by using a set of per-option read/write locks, called "option locks". A read acquisition of an option lock indicates that a thread is currently using that option (or some value derived from it). Updates to a configuration option must first acquire the option's lock in write mode. If the option being updated is in use, the write acquisition will stall until all outstanding uses of the option finish and all read holds are released. The option write lock is immediately released after update is effected. The programmer is not responsible for acquiring the write locks during configuration update: all updates are delegated to our implementation's runtime which handles locking.

Read-acquisition of option locks is automatically performed by Staccato during a staleness check. Recall that staleness checks occur after every value read. After a value is read, but before the staleness check is performed, Staccato read-locks the option locks for every option in the read value's configuration history. These locks are held in read mode after control returns to the host program. The scope of a value's use is approximated as the method containing the initial read of that value. Thus, the read-acquisitions on the option locks acquired during a read are held until the containing method completes. Computing the scope precisely is not possible, so we use end-of-method as heuristic. However, using our heuristic was sufficient to prevent several errors that Staccato would have otherwise occurred during our evaluation.

```
1 class RequestManager implements StaccatoFieldRepair {
2   String targetIp, apiKey;
3   String doRequest() {
4      Request req = new ApiRequest(targetIp);
5      req.send(apiKey);
6      return req.response();
7   }
8   Object _StaccatoRepairField(String fieldName,
9     Object oldValue, Exception staleException) {
10    if(fieldName.equals("apiKey")) {
11      return apiKey = Config.get("api-key");
12    } else if(fieldName.equals("targetIp")) {
13      return targetIp = Config.get("target-ip");
14    } else { throw staleException; }
15  }
16 }
```

Figure 7 An updated version of the code in Figure 1 which uses Staccato repair callbacks. The `Reloadable` interface has been replaced with the `StaccatoFieldRepair`. The new method, `_StaccatoRepairField` is called when Staccato detects that one of the fields is stale. The updated value returned from the method automatically replaces stale field.

4.2 Value Repair

Staccato's value repair is an extension to the staleness checking mechanism. Unlike schedule repair, value repair is not fully automatic: we require the programmer to provide a repair function that updates a stale value to reflect the latest version of the configuration. Although Staccato can automatically *detect* a stale value, automatically *updating* it to the correct value is beyond pure automation because how to respond in the presence of stale data is fundamentally application-specific.

Staccato's value repair operates exclusively on object fields. Staccato already intercepts all field reads for checking. When the Staccato runtime detects that a value read from a field is stale, it checks if the object hosting the field implements an interface that provides the Staccato update callback. If the callback exists, Staccato calls it with the name and current value of the stale field. The callback may rebuild the stale field in an application defined way. The updated field value is then returned from the hook and transparently replaces the old, stale value as the result of the original field read. If the callback is unable to repair a field, the original error is reported as usual; failure to repair a field is not itself an error. Figure 7 shows a simplified application of this update mechanism to the example given in Figure 1 in Section 2.

Repairing (or updating) a stale configuration derived value will likely use the same options used to construct the old value. As a convenience, Staccato calls the update callbacks while holding the option locks described in Section 4.1 in read mode. Thus, although update callbacks may execute in a multithreaded context, the callback effectively sees a consistent, immutable view of the necessary configuration options during the rebuild operation. As a further convenience, Staccato ensures that no threads concurrently execute the update callback on the same object.

5 Implementation

We have implemented our technique in a prototype tool named Staccato. Staccato is an offline bytecode instrumentation tool for Java programs. Staccato modifies a program's bytecode to support tracking configuration histories and also inserts code to perform check-

ing and propagation. The configuration tracking of STACCATO is built on top of a modified version of the Phosphor tool by Bell and Kaiser [4]. STACCATO also includes a runtime library that implements the operations described in Section 3. STACCATO does not automatically integrate with the program's configuration abstraction, this must be performed by the programmer when initially integrating STACCATO into the software (see Section 5.3).

5.1 Basic Operation

Applying STACCATO is a two-step process. First, our modified version of Phosphor instruments the source program to add information tracking for primitive types. In a second pass, STACCATO uses ASM [19] and Javassist [7] to add code that calls into the STACCATO runtime to perform the check and propagation operations described in Section 3. For each method covered by a correctness condition, STACCATO instruments all array, variable, and field reads, as well as method calls to check read values against the correctness condition selected for the method. STACCATO also inserts code at the end of methods annotated with `@StaccatoPropagate` to perform history propagation. Phosphor already adds unconditional propagation from source values to outputs for primitive instructions such as integer addition; we replaced the merge operation used by Phosphor for these instructions with a call into our runtime library.

STACCATO does not require a modified JVM. To ensure information is soundly tracked through calls to the Java Class Library, a program must use a version of the JCL that has been instrumented by Phosphor. To ease integration, we also added propagation to certain "primitive" operations in the JCL, such as string concatenation or string-to-integer parsing. To use STACCATO, a program must be launched with a special JVM flag that adds our instrumented versions of the system classes to the system classpath.

5.2 Tracking Configurations

To reduce memory overhead, STACCATO does *not* unconditionally instrument every type to carry configuration history. STACCATO uses the programmer's `@StaccatoPropagate` annotations to avoid instrumenting types that will provably never carry configuration information. For a type that *may* carry configuration information, STACCATO adds a special field to the class definition. STACCATO also modifies the class definition to implement an interface that marks the class as carrying configuration information. This interface exposes two methods that provide the functionality to get and set the hidden field. STACCATO synthesizes these methods and inserts them into the class definition. For primitive values (which do not have fields), we reuse the shadow taint variables added by Phosphor.

We chose the interface approach over using reflection for two reasons. First, determining if a type carries configuration information reduces to a relatively inexpensive `instanceof` check. The second reason is to integrate with other dynamic bytecode rewriting tools. There are several libraries that perform instrumentation at runtime to introduce features like database connection pooling.[4] These tools introduce wrapper classes that encapsulate the original configuration carrying values. This hiding makes access to a hidden tag field via reflection impossible. However, we found that these tools often make an effort to preserve the interfaces of types being instrumented. This preservation is done at the type level (a wrapper class for some type T also implements the same interfaces as the original type

[4] e.g., `http://proxool.sourceforge.net/`

T) and in terms of the behavior of the generated class (by delegating method calls to the wrapped value). Using interfaces allowed Staccato to seamlessly integrate with these dynamic tools. There is no guarantee of interoperability but we found that using interfaces worked with all the dynamic tools we encountered.

5.3 Host Program Integration

Staccato does not manage the configuration of the software being analyzed: it is the responsibility of the programmer to integrate the Staccato runtime with the software's configuration abstraction. Staccato assumes that the software uses a key-value data structure for configuration information. The key-value abstraction is widely used in practice for storing configurations, including the Java Properties API, the Windows Registry, and several real-world software projects [33, 16]. All of the software configurations in our evaluation used this abstraction. Rabkin et al. have shown that complex, hierarchical configurations can be adapted to the key-value model [33]. We also assume that all configuration values are strings. In principle, our approach and implementation could be extended to support configuration values of arbitrary types. In practice, all programs we encountered used strings for storage and performed parsing/serialization of these strings where necessary.

Integrating a program's configuration with Staccato requires only minimal source changes. The programmer must delegate all set, get, and delete operations on the configuration key-value store to the Staccato runtime. For example, suppose a program's configuration abstraction is stored in a `HashMap` variable named `conf`. Delegating get operations to Staccato involves changing calls of the form `conf.get(key)` to `Staccato.get(conf, key)`. Similar changes are made for delete and set operations. The delegation ensures that Staccato increments option epochs on configuration update and correctly tags configuration values read by the program. The API exposed by the Staccato runtime is very general; we were able to incorporate all configuration abstractions we found in our evaluation.

5.4 Propagation Coordination

Multithreaded execution introduces the possibility for two or more propagation operations to occur concurrently. If two or more propagation operations involve the same object concurrently, the Staccato runtime uses locking to impose an *arbitrary* order on configuration history propagation. The history merge operation (see Sections 3.2 and 3.4) is commutative: two or more propagation operations with the same target object can occur in any order without changing the final configuration history associated with the object.

Adding locking to arbitrary user programs risks introducing deadlocks, so we modeled the locking protocol of Staccato in the Alloy model finder [15]. Our verification of Staccato's locking was bounded, but we used significantly large parameters to achieve high confidence in our results. Using this model, we verified that the locks introduced by Staccato do not deadlock. Staccato's locks can interact with a program's existing locks to produce deadlocks. This is because a thread of execution may hold several option locks in read mode, and perform arbitrary lock acquisitions. In practice, this was never a problem in our evaluation.

5.5 Polymorphism and Subtyping

Staccato does not instrument every read within a method, only those it cannot prove will not carry configuration information. During instrumentation, Staccato may encounter

a value of type T that does not itself track configuration information, but there is some subtype of T that does. In this scenario, STACCATO cannot statically determine if checking should be performed. STACCATO handles this ambiguity by inserting *conditional* checks. If the runtime type of an object does not carry configuration information, a conditional check is a no-op, otherwise a regular check is performed.

Java generics pose a similar problem. Java implements polymorphism using type erasure [5]: `Object` is used to represent type variables at the bytecode level. As every type is a subtype of `Object`, the use of type variable in a target program will lead to ambiguity for STACCATO's instrumentation. However, this situation is simply a degenerate case of the subtyping ambiguity described above. STACCATO therefore resolves all ambiguity caused by generics by exclusively using conditional checks.

5.6 Limitations

STACCATO inserts the code that performs the propagation operation at method return after the method body has completed. The propagation target's object's configuration history will therefore be updated *after* the object state has been changed. As a result, the propagation is not necessarily atomic with the body of the method.[5] This can result in the configuration history for an object briefly not reflecting the current state of the object. Unfortunately, it is impossible for STACCATO to automatically incorporate the propagation operation into a synchronization scheme for a method. This limitation did not prevent us from effectively finding bugs and did not admit any false positives.

The association of configuration histories with program values requires that two logical values with distinct configuration histories must have unique identities. This restriction means that STACCATO interacts poorly with memoization or singleton objects, as two or more logically separate configuration histories may be conflated in the same program value. This was primarily a problem for literal strings and enumeration variants. All occurrences of the same string literal and enumeration variant within a single JVM instance have the same object identity. For literal strings, boolean options were particularly problematic: boolean configuration values were almost always stored using the literals `"true"` and `"false"`. STACCATO conservatively performs a deep-copy when it detects it would otherwise propagate configuration history to a string literal or enumeration.

However, copying enumerations will break a program that relies on the referential equality between two enumeration variants with the same name. To work around this, unboxing operations are added to equality tests that involve enumerations. Unboxing restores a copied enumeration back to the original singleton object for purposes of comparison. This unboxing imposes a non-trivial overhead, and the programmer must opt-in to this mechanism.

Finally, incorrect annotations by users can produce incorrect analysis results. Over-annotating can lead to false positives. However, these false positives reveal where the programmer's assumptions (expressed in his/her annotations) about how a program handles DCU are incorrect. Under-annotation results in bugs being missed, but existing analyses that track the flow of configuration values, e.g. Lotrack [20], and ConfAnalyzer [32], can help guide the user to the correct annotations. This limitation is inherent and would require a user-study to measure the extent of the problem in practice.

[5] If the method is marked as `synchronized`, the propagation is protected by the object monitor that protects the entire method body.

6 Evaluation

Having defined our approach to finding bugs in dynamic software configuration, we present our evaluation of STACCATO's effectiveness. In evaluating STACCATO, we were interested in the following 4 questions:

1. Does software with dynamic configurability have violations of our correctness conditions?
2. How effective is STACCATO at finding these errors?
3. For some real applications, what is the annotation burden of using STACCATO?
4. What is the performance impact of using STACCATO?

We focused our evaluation on open-source applications with high-levels of run-time configurability. We chose three projects of substantial size but approachable complexity: Openfire (version 3.9.3),[6] a full featured chat server that implements the XMPP IM protocol, JForum (version 2.1.8),[7] a widely deployed forum software, and Subsonic (version 5.2.1),[8] a music streaming server. Each project has many configuration options and extensive concurrency.

We did not perform an exhaustive evaluation on Solr as the code-base was close to 500,000 SLOC: considerably larger than our next largest evaluation target (Openfire). No technical limitation prevents using STACCATO on Solr: the instrumentation process and dynamic analysis scale independently of code-base size. However, using STACCATO requires understanding a code-base and 500,000 lines felt beyond what we could reliably do ourselves. We did perform an informal evaluation on Solr where we tried to detect the bug mentioned in the introduction using STACCATO. We needed only a handful of propagation annotations and one check annotation to re-find the bug. We believe, but haven't substantiated, that a team familiar with Solr could use STACCATO without undue burden.

Experimental setup

We manually annotated each application after first becoming familiar with the code-base and how the software uses configuration values. We also integrated each software's configuration abstraction with the STACCATO runtime.

We were unable to find extensive functional tests for any of the projects. We developed our own functional tests for each of the three software projects. Due to the differences in the software under evaluation, we had to use different evaluation techniques depending on the software. For Openfire, we used Tsung[9] version 1.4.2. For JForum and Subsonic we used Apache JMeter[10] version 2.12. We developed test plans in these tools that exercised core functionality of each software (these tests are included in the STACCATO distribution). Each functional test client executes in a loop. On each iteration of the loop, a test clients sleeps for a short, randomly selected period of time and then performs a randomly selected test action. These test actions were prepared by us and were designed to use one piece of the core functionality of the software under test. For example, one of the test actions for JForum involves sending a private message from one user to another.

To induce configuration errors, we also developed a *havoc mechanism*. This havoc mechanism consists of several test clients that execute alongside the functional test clients. Each

[6] `http://www.igniterealtime.org/projects/openfire/index.jsp`
[7] `http://jforum.net/`
[8] `http://www.subsonic.org/pages/index.jsp`
[9] `http://tsung.erlang-projects.org/`
[10] `http://jmeter.apache.org/`

havoc client executes in an infinite loop. During each iteration of the loop, the havoc client sleeps for a short, randomly selected period of time and then performs a mutation to the software's configuration. This mutation was done by simulating an HTTP request to the administrative webpages of the software under test. These administrative webpages also validated our havoc updates (e.g., by rejecting attempts to select negative port numbers). The havoc clients were also restricted to choosing mutations that we had manually prepared in consultation with the program code and documentation. Any errors reported by Staccato during testing were logged for collection. We report our findings from these experiments in Section 6.3.

We evaluated the performance impact of Staccato along two dimensions, slowdown and memory overhead. To calculate the slowdown, we ran each software's test suite on the instrumented and uninstrumented versions 5 times each and measured response times. Before collecting any data we ran a shorter version of each project's test suite to control for the effects of JIT compilation and application-specific startup actions. We disabled the havoc mechanism during performance testing, as the high rate of configuration updates (several times per second) is not realistic for measuring performance. To measure the memory overhead of Staccato, we ran the same test suites (again with a brief warmup period) and used Java's JMX technology to monitor the program's memory usage. We sampled the JVM's reported memory usage at one second intervals. Before taking each measurement we triggered a garbage collection.

All experiments were run on a Dell Latitude E-5440 with a 4 core Intel Core i5 processor at 2.00 GHz and with 16GB of RAM. We used version 1.7.0-91 of the OpenJDK JVM.

6.1 Summary

The annotation burden of using Staccato is extremely low: the ratio of annotation to SLOC is below 1%. Even including changes to integrate with Staccato and update handlers, we found that the effort needed to use Staccato in a project is minimal. Further, Staccato was able to accommodate most configuration options of interest and usage patterns in the projects that we evaluated.

We found DCU errors in all of our evaluation targets, despite widely different approaches to configuration discipline and multithreading. Many of the DCU errors that we found were violations of the staleness condition caused by two concurrent configuration updates. Two of the projects we evaluated with Staccato also had DCU errors that could occur in a single-threaded context, indicating that DCU errors are not just the result of insufficient testing of multithreaded software. We encountered only one false positive during our analysis.

Staccato imposed a moderate performance penalty in our tests; we measured a maximum overall slowdown of 5.30x and a memory overhead of at most 144.40%. This is overhead low enough to use Staccato as a bug-finding tool in pre-deployment. Our experience suggests combining Staccato with an automated havoc test like those used in our evaluation can be an effective technique for finding DCU errors. Staccato's overhead makes it unlikely that the repair mechanisms described in Section 4 can be used in production. However, a developer can use Staccato's repair mechanisms to rapidly develop DCU implementations or fixes. From an initial implementation using Staccato, a programmer can develop a more efficient manual solution.

Table 1 Counts of lines changed to integrate with STACCATO. **Annot.** counts explicit check and propagation annotations. **Flow** are changes to calling conventions, the introduction of helper methods, or the use of wrapper classes for integration with STACCATO. **Repair CB** are repair callback code. **Bug-Fixes** are fixes for concurrency bugs in the project. These consist primarily of field accesses that were not well ordered according to the Java Memory Model [26]. **Conf-Abs.** are changes to integrate the project's configuration abstraction with the STACCATO runtime. **SLOC** counts the total source lines of code in the original, uninstrumented project.

Project	Annot.	Flow	Repair CB	Bug-Fixes	Conf-Abs.	Total	SLOC
Openfire	100	184	451	78	212	1,025	85,416
JForum	11	44	13	2	29	99	29,568
Subsonic	13	52	0	0	118	183	29,592

6.2 Integration Effort

As a proxy for programmer effort, we measured the number of line changes necessary to integrate STACCATO with each of our evaluation targets. We count explicit propagation and checking annotations as well as lines changed to integrate the software's configuration abstraction with STACCATO's runtime, changes to calling conventions to ease application of annotations, and repair callbacks. The breakdown of lines changed is shown in Table 1. The ratio of annotations to total lines for each project is 0.12%, 0.04%, and 0.04% for Openfire, JForum, and Subsonic respectively. Including all source lines changed gives: 1.20%, 0.33%, and 0.62% for Openfire, JForum, and Subsonic respectively. The relatively high percentage for Openfire can be attributed to large number of lines added for update callbacks (column **Repair CB** in the table).

Qualitatively, most of our effort was spent understanding each code-base. A team familiar with an application would be spared this effort. Adapting a program once we understood how its dynamic configuration worked was largely formulaic. Many bugs were found with no extra annotations as many methods are checked by default against the consistency condition. Finding staleness violations required some manual annotation. However, recall that annotations to control checking by STACCATO can be applied to an entire class. Thus, when we added annotations for finding staleness violations, no pre-existing knowledge of exact bug location was required, only an intuition that a class encapsulated some persistent, configuration-derived state. Finally, we found most candidates for propagation annotations simply by inspecting configuration access sites and determining what methods were called with the newly returned configuration values.

Checking Coverage

We found that the configuration model and primitives exposed by STACCATO were sufficient to achieve good coverage of options checked in each evaluation target. Our functional tests exercised between 25%–86% of options in our evaluation programs. STACCATO was able to track and check all options exercised by our tests.

We measured coverage by modifying the STACCATO runtime to record the options checked by STACCATO and the correctness condition being checked. We also recorded which options were read or updated during test execution. Options not involved in at least one update operation are not counted for coverage purposes. Uses of these options were trivially validated by STACCATO but are uninteresting for evaluation purposes. Options that were updated but never re-read *are* included; STACCATO still validated that old copies of the option were used consistently and did not cause violations of the consistency condition.

Table 2 Classification of options available in each application and coverage of our functional test suite. **Checked** counts options checked during our tests. **Update** are options updated using STACCATO's repair mechanism. The next 3 columns count options not included in our tests. **Imm.** are immutable and therefore uninteresting for our evaluation. **Int.** are internal, undocumented options for which we lacked sufficient domain knowledge of the software. **Other** are options that were untestable because of missing proprietary technology or update mechanisms with unfixable concurrency bugs. **Untested** are all remaining options not exercised by our functional tests. **Cov** measures the percentage of options tested in our evaluation.

Project	Checked	Update	Imm.	Int.	Other	Untested	Total	Cov.
Openfire	24	21	68	73	4	61	251	25.14%
JForum	37	1	3	0	0	6	47	86.36%
Subsonic	33	0	0	0	0	15	48	68.75%

Table 3 Counts of errors found by STACCATO. **Stale Read** are violations of the staleness condition. **Incons. Read** are violations of the consistency condition. **Incons. CF** are violations of the consistency condition for control-flow.

Project	Stale Read	Incons. Read	Incons. CF
Openfire	11	0	0
JForum	0	5	0
Subsonic	1	2	1

In all tests, every configuration option mutated by the havoc mechanism was checked at least once. This result indicates that our approach accurately models the configuration updates of the evaluation programs. Across all projects, most options were checked against the consistency condition rather than the staleness condition. However, all programs used checks for staleness at least once. Openfire and Subsonic both used the condition to find bugs and JForum used staleness checking in combination with repair to add support for configuration updates.

Our tests exercised only a representative set of options for each program in our evaluation. There is no restriction other than annotation and test development effort preventing 100% coverage of options. We felt that achieving full coverage in our tests would provide low marginal benefit for the amount of effort required. For example, we did not test the configuration options that control Openfire's operation in multi-server clusters. An overview of which options we checked during our evaluation is presented in Table 2. We calculate the coverage of our tests as:

$$\frac{Checked + Update}{Checked + Update + Untested + Internal}$$

Coverage for our evaluation set is 25.14% 86.36% and 68.75% for Openfire, JForum, and Subsonic respectively. Openfire's relatively low coverage is due to the high number of options available in that program. Some of these options (counted in column Untested) controlled functionality that was difficult to test automatically or that required complex test environments (e.g., mutli-server cluster options). We also excluded many undocumented options available in Openfire that cannot be set via normal means (counted in column Internal).

```
1 public void setLogDir(String directory) {
2   Config.set("audit.log.dir", directory);
3   this.auditDir = directory;
4 }
```

Figure 8 Simplified atomicity bug during configuration update. If two threads concurrently execute `setLogDir`, the value in the `auditDir` field may not be the most recent version of `"audit.log.dir"` option. This violates the object's implicit invariant. STACCATO detects this error on subsequent reads of the `auditDir` field.

```
1 static Map<String, ContactList> cache = ...;
2 ContactList getContactList(String user) {
3   if(cache.containsKey(user)) {
4     return cache.get(user);
5   } else {
6     ContactStorage store = getContactStorage();
7     cache.put(user, new ContactList(user, store));
8     return list;
9   }
10 }
11 ContactStorage getContactStorage() {
12   String backend = Config.get("contact.classname");
13   /* reflectively instantiate the backend specified by
14   "contact.classname" */
15 }
```

Figure 9 Simplified example of contact list bug in Openfire. A stale version of the backend specified by `"contact.classname"` will persist in ContactStorage objects cached in `cache`.

6.3 Effectiveness

Table 3 summarizes the errors that we found in our three evaluation targets. In every evaluation target, STACCATO found multiple dynamic configuration update errors. In total, STACCATO discovered 20 errors across the three evaluation programs. The most common bugs are staleness violations that occur when two concurrent configuration updates were performed. However, two of the three projects also contain errors that do not depend on concurrency to manifest. We now discuss some example bugs found in each evaluation target.

6.3.1 Openfire

All of the bugs found in Openfire were violations of the staleness condition. The most common error in Openfire was reads of out-of-date configuration data caused by non-atomic configuration updates. Figure 8 shows a simplified example of the problematic configuration update idiom behind these errors.

We also found a DCU error in Openfire that does not require a specific thread schedule. The server administrator may configure, at runtime, the implementation used to store a user's contact list. Each user's contact list object holds a reference to the storage backend specified by the administrator. STACCATO found that contact lists loaded before a change to the storage backend would continue to use the old backend object. This could cause changes made to a user's contact list to be lost. The underlying defect was caused by the contact list objects being cached in a static field. A sketch of this scenario is shown in Figure 9.

Using the repair callback mechanism described in Section 4, we were able to introduce dynamic updates for 21 options. We also added repair for 2 of the errors that we found in

our evaluation. The locking guarantees provided by STACCATO allowed us to write update callbacks that focused only on implementing the actual configuration update and did not require us to add any extra locking. STACCATO also ensured that we did not introduce any new dynamic configuration update errors when adding this new DCU functionality. We validated that our update code was correct by extending the havoc mechanism to include changes to the configuration options for which we wrote update callbacks; no errors were reported by STACCATO in any of the options we tested in this way.

6.3.2 JForum

JForum maintains less configurable global state than Openfire, so STACCATO did not find any violations of the staleness condition. However, STACCATO found 5 violations of the consistency condition.

Most of these errors were the result of two or more reads of the same configuration option without any synchronization. For instance, when generating a response, JForum reads an option twice to set two separate response parameters. The two reads are not protected by any synchronization, and STACCATO detected that a concurrent update of the option would yield an inconsistent response. The single false positive found during our evaluation occurred under similar circumstances. JForum reads a single option twice (again, without synchronization), except that both reads populate the *same* response parameter. STACCATO was unable to detect that the second write overwrote the first, concealing any inconsistency.

One of the consistency errors STACCATO discovered caused JForum to mislabel the encoding used in an HTTP response. Finding this error involved tracking object dependencies at a level not possible with just primitive value dependencies. The bug is caused by the configuration option that controls response encoding being read twice during request handling: once in the method that generates the HTML response, and in another method that sets the response headers. Using to `@StaccatoPropagate` annotations, STACCATO was able to capture the dependencies between the encoding option, the generated output, and the response headers.

STACCATO also found a consistency error that does not rely on a specific thread schedule to manifest. JForum pre-computes the URLs of emojis available on the forum and stores the results in a cache. The emoji URLs are built in part using the URL of the forum, which is configurable. However, after the forum URL is updated, the emoji URLs in the cache are not updated. The forum URL is also read on each request to render the header and footer content of each page. STACCATO detected that after the forum URL is updated, responses that include emojis contain two inconsistent versions of the forum URL.

The handful of options that control persistent state all require a restart to take effect, indicating a conservative approach on the part of the JForum developers towards configuration updates. We were able to use STACCATO's repair functionality to transparently and safely introduce online updates for one of these options.

6.3.3 Subsonic

Compared to JForum and Openfire, Subsonic has little configurable global state. The one exception was in the LDAP integration component. Subsonic caches its connection to an LDAP server in a static field and rebuilds the connection when it detects that the LDAP configuration has changed. A simplified form of this update algorithm is shown in Figure 10. However, although the `getConnection` method is synchronized, the method does not synchronize with updates to the software configuration. As a result, `lastChecked` may hold the most

```
static LdapConnection ldapConnection = ...;
static Time lastChecked = ...;
synchronized LdapConnection getConnection() {
  if(lastChecked < Config.lastUpdateTime()) {
    ldapConnection = Ldap.connect(Config.get("ldap-config"));
    lastChecked = Config.lastUpdateTime();
  }
  return ldapConnection;
}
```

Figure 10 A sketch of the Subsonic LDAP connection staleness bug. In this example `Config.lastUpdateTime()` returns the time of the most recent update to the configuration. Although the update code uses `synchronized`, the update still contains an error. If an update to the `"ldap-config"` option occurs between lines 5 and 6, the program incorrectly concludes that the `ldapConnection` field is up-to-date.

recent update time for the configuration but the `ldapConnection` could contain a connection built with an old version of the `"ldap-config"` option. Notice that this idiom is very similar to the value repair scheme described in Section 4.2, although the option locks (Section 4.1) prevent a smiliar error from occurring within our implementation.

Subsonic also contains a control-flow consistency error. A method that controls the music play queue iterates over a list of playlist items and on each iteration checks an option that determines if Subsonic is running behind a firewall. If this option is set, extra processing is performed on the playlist item. STACCATO flagged that if the option was changed during an iteration of the loop, Subsonic would inconsistently process the entire list as a result.

One final example bug found by STACCATO was caused by one logical option (the server locale) being stored across three different configuration options. The reads of the three options where not synchronized with the update mechanism of the options. As a result, if a read of the three locale options occurred concurrently with an update, STACCATO detected that the user would receive an invalid locale (e.g, the US version of Japanese).

6.4 Performance Impact

Finally, we measured the performance impact of STACCATO on our evaluation targets. We measured two metrics: memory overhead and overall slowdown. Across all evaluation programs, the largest overall slowdown (averaging over test actions) was 5.30x and memory overhead was below 2x. A graph of the slowdown results can be found in Figure 11.

We calculated overall slowdown as the geometric mean of the ratio between average response times with and without Staccato, after excluding the largest 5% of response times as outliers. Tsung (the technology used to test Openfire) did not provide individual response times, so we did not filter outliers when calculating Openfire's slowdown. The number and duration of the outliers was broadly similar with and without STACCATO, with high variance across runs.

The overall slowdown for the projects was 1.26x, 5.30x, and 2.11x for Openfire, JForum, and Subsonic respectively. This is generally competitive with other dynamic analyses and is consistent with the performance reported for the Phosphor tool [4] on top of which STACCATO is built.[11] The majority of extra time is spent combining configuration histories for

[11] The slowdown reported in the Phosphor paper is for a version that supported only integer valued tags. STACCATO is built using a more recent version that supports arbitrary tag types, which is necessarily slower.

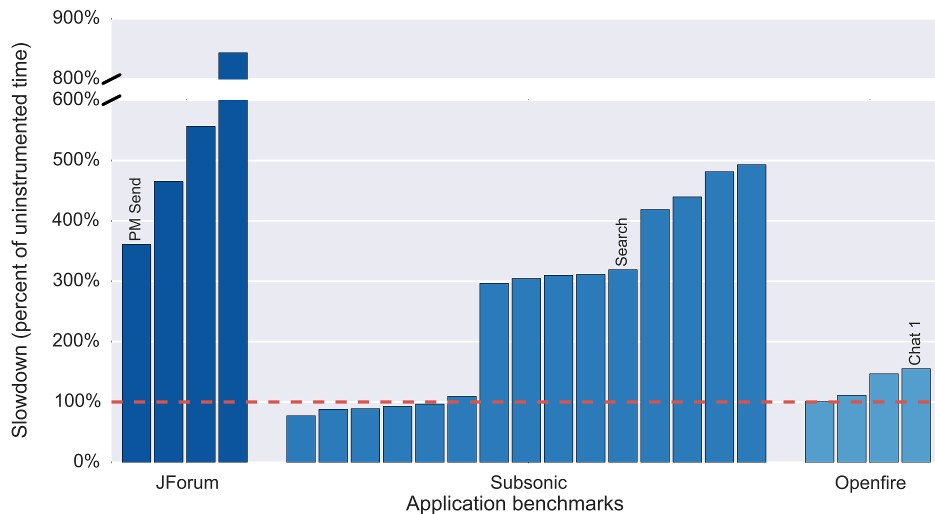

Figure 11 The average slowdowns for the evaluation applications. Each individual bar represents the average slowdown of a single test action in the test suites we created for the project. For example, the "PM Send" bar measures the average slowdown experienced when sending a private message to another user on the forum. Similarly, the "Search" and "Chat 1" bars respectively measure the average slowdown for searching Subsonic's music catalog and sending a sequence of chat messages on the Openfire server. The overall slowdown for each application is computed by taking the geometric mean of these average slowdowns.

primitive types. History merging is performed via a relatively expensive method call that is inserted after every arithmetic, floating point, and boolean operation. In the common case, the configuration histories being merged are empty. Although we optimized our runtime heavily for this common case, STACCATO runs into the inherent overhead of method calls. STACCATO could be combined with a whole-program static analysis to prune merge operations when the configuration histories involved are provably empty.

The memory overhead of STACCATO was: 110.83%, 144.40%, and 114.43% for Openfire, JForum, and Subsonic respectively. For each application, we calculated the memory overhead by averaging the observed heap sizes during the instrumented test runs, and similarly for the uninstrumented test runs. We took the ratio of these two averages to be the total memory overhead. Most of this memory overhead is added by Phosphor for shadowing primitive variables as noted in the Phosphor paper. Our memory overhead is generally lower than that reported for Phosphor as we do not add tag fields unconditionally to every object as explained in Section 5.2. We expect that memory usage could be improved with a static analysis to remove provably empty shadow state.

7 Related Work

Configuration management is an active area of research [42, 39, 40, 1, 2, 16, 43, 9, 34, 27, 20, 33, 41]. To our knowledge, no existing research has examined the problem of DCU errors. Some existing work has used approaches similar to ours to study different problems. ConfAid [3] uses a dynamic information-flow analysis similar to ours to diagnose software misconfigurations. It associates each program value with a *configuration set*: this set tracks which configuration options potentially influenced the construction of the value. This is very similar to STACCATO's configuration histories. However, ConfAid reasons only about static configurations, and does not track versions like STACCATO. Rabkin et al. [32] also

tracked configuration values for the purposes of diagnosing configuration errors. However, their analysis does not reason about dynamic configuration updates, and uses a conservative static analysis.

Staccato's analysis builds on extensive work on dynamic information-flow analysis [11, 12, 4]. Our approach precisely tracks data dependence, but handles handles dependencies from indirect flow using a heuristic. Dynamic analyses that precisely track dependencies from control-flow do exist (e.g., [18, 8]) but suffer scalability that limits their application to large programs such as the ones we use in our evaluation. However, given a scalable framework for control-flow tracking, Staccato could be extended to support control-flow.

Several of the DCU errors we found were the result of atomicity violations, such as the JForum response parameter bug, or the example in Figure 8. These errors are similar to linearizability errors. There is research into automatically checking linearizability [35, 6, 36] and repairing operations that are not linearizable [23]. However, existing linearizability research focuses on linearizability of ADT operations and cannot handle the arbitrary operations performed on configuration options. Current work on dynamic atomicity checkers [10, 38] offer another possible solution to detect consistency violations involving configurations. However, linearizability and atomicity checkers would not find DCU errors that involve our staleness condition.

Staleness violations result from a DCU invalidating one thread's assumptions about the (global) configuration state. This is similar to check-then-act errors [22, 21, 29] that result from a concurrent update invalidating assumptions established by a check operation. In contrast to check-then-act errors, Staccato's consistency condition allows concurrent updates to the global configuration, provided that a thread never observes inconsistencies.

Finally, there has been considerable research in the field of dynamic software updates (DSU) [24, 13, 30, 25, 14, 31, 28]. DSU attempts to introduce arbitrary code changes to running software without service interruption. Dynamic configuration updates can be viewed as a specific case of DSU: each configuration update is a controlled change to running software at runtime. However, existing DSU research does not provide checking for bugs in a software's current dynamic configuration mechanism, and offers only a potential alternative for current DCU mechanisms. Unfortunately, even in the state of the art, existing DSU techniques require significant effort on the part of the programmer to integrate with existing applications and impose non-trivial performance overhead. One interesting common point between existing DSU research and the approach in Staccato is how to maintain consistent application state in the presence of an online update. Many existing approaches require the programmer to write update hooks (e.g., [28]): this is similar to the approach taken by Staccato for program repair (Section 4.2).

8 Conclusions and Future Work

This paper presented the first study of errors in dynamic configuration updates. We attacked the problem of diagnosing incorrect updates with a dynamic analysis tool Staccato, which detects stale or inconsistent views of the software configuration. We evaluated Staccato on three open-source projects and found bugs in all three. Staccato imposes moderate performance overhead, and requires only low manual annotation overhead. In future work, we hope to complement Staccato with static analysis to lower the need for test cases to find DCU errors.

Acknowledgments. The authors thank James Bornholt, Doug Woos, Pavel Panchenkha, James Wilcox, and Emina Torlak for helpful discussions and feedback on early drafts of

this paper. We would also like to thank Jonathan Bell for his help with the Phosphor tool and adding support for enumerations. Finally, we thank the anonymous reviewers for their helpful feedback.

References

1. Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.
2. Mona Attariyan and Jason Flinn. Using causality to diagnose configuration bugs. In *USENIX Annual Technical Conference*, 2008.
3. Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.
4. Jonathan Bell and Gail Kaiser. Phosphor: illuminating dynamic data flow in commodity JVMs. In *OOPSLA*, 2014.
5. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*.
6. Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. In *PLDI*, 2010.
7. Shigeru Chiba. Javassist-a reflection-based programming wizard for Java. In *OOPSLA Workshop on Reflective Programming in C++ and Java*, 1998.
8. James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA*, 2007.
9. Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with iTuned. *Proceedings of the VLDB Endowment*, 2009.
10. Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, 2008.
11. Malay Ganai, Dongyoon Lee, and Aarti Gupta. Dtam: dynamic taint analysis of multi-threaded programs for relevancy. In *FSE*, 2012.
12. Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Computer Security Applications Conference*, 2005.
13. Christopher M Hayden, Karla Saur, Michael Hicks, and Jeffrey S Foster. A study of dynamic software update quiescence for multithreaded programs. In *Workshop on Hot Topics in Software Upgrades*, 2012.
14. Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. In *OOPSLA*, 2012.
15. Daniel Jackson. Software abstractions: Logic. *Language, and Analysis. MIT Press*, 2012, 2006.
16. Dongpu Jin, Myra B Cohen, Xiao Qu, and Brian Robinson. PrefFinder: getting the right preference in configurable software systems. In *ASE*, 2014.
17. Dongpu Jin, Xiao Qu, Myra B Cohen, and Brian Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *ICSE*, 2014.
18. Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.
19. Eugene Kuleshov. Using the ASM framework to implement common Java bytecode transformation patterns. *Aspect-Oriented Software Development*, 2007.
20. Max Lillack, Christian Kästner, and Eric Bodden. Tracking load-time configuration options. In *ASE*, 2014.
21. Yu Lin. *Automated refactoring for Java concurrency.* PhD thesis, University of Illinois at Urbana-Champaign, 2015.

22 Yu Lin and Danny Dig. Check-then-act misuse of Java concurrent collections. In *ICST*, 2013.
23 Peng Liu, Omer Tripp, and Xiangyu Zhang. Flint: fixing linearizability violations. In *OOPLSA*, 2014.
24 Kristis Makris and Rida A Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *USENIX Annual Technical Conference*, 2009.
25 Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *EuroSys*, 2007.
26 Jeremy Manson, William Pugh, and Sarita V Adve. *The Java memory model*, volume 40. ACM, 2005.
27 Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *ICSE*, 2014.
28 Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *PLDI*, 2006.
29 Mathias Payer and Thomas R. Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *VEE*, 2012.
30 Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for Java on a stock JVM. In *OOPSLA*, 2014.
31 Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. Javadaptor—flexible runtime updates of Java applications. *Software: Practice and Experience*, 2013.
32 Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *ASE*, 2011.
33 Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *ICSE*, 2011.
34 Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, 2010.
35 Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA*, 2011.
36 Ohad Shacham, Eran Yahav, Guy Golan Gueta, Alex Aiken, Nathan Bronson, Mooly Sagiv, and Martin Vechev. Verifying atomicity via data independence. In *ISSTA*, 2014.
37 Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: Improving configuration management with operating system causality analysis. In *SOSP*, 2007.
38 Liqiang Wang and Scott D Stoller. Runtime analysis of atomicity for multithreaded programs. *Transactions on Software Engineering*, 2006.
39 Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. *Science of Computer Programming*, 2004.
40 Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating range fixes for software configuration. In *ICSE*, 2012.
41 Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *SOSP*, 2013.
42 Sai Zhang and Michael D Ernst. Automated diagnosis of software configuration errors. In *ICSE*, 2013.
43 Sai Zhang and Michael D Ernst. Which configuration option should I change? In *ICSE*, 2014.