Precomputing Possible Configuration Error Diagnoses

Ariel Rabkin and Randy Katz
EECS Department, UC Berkeley
Berkeley, California, USA
{asrabkin,randy}@cs.berkeley.edu

***Abstract*—Complex software packages, particularly systems software, often require substantial customization before being used. Small mistakes in configuration can lead to hard-to-diagnose error messages. We demonstrate how to build a map from each program point to the options that might cause an error at that point. This can aid users in troubleshooting these errors without any need to install or use additional tools. Our approach relies on static dataflow analysis, meaning all the analysis is done in advance. We evaluate our work in detail on two substantial systems, Hadoop and the JChord program analysis toolkit, using failure injection and also by using log messages as a source of labeled program points. When logs and stack traces are available, they can be incorporated into the analysis. This reduces the number of false positives by nearly a factor of four for Hadoop, at the cost of approximately one minute's work per unique query.**

I. Introduction

Studies have shown that human administrators almost inevitably make mistakes, even when given explicit step-by-step directions [1]. Given an unfamiliar program, errors and fumbling are inevitable. For widely-used programs, web search can bring up user reports of problems and solutions. But configuration debugging is still a thorny problem for specialized open-source programs. Projects can accumulate configuration options that were useful for solving some particular problem at a particular site at some point in time. Without centralized management, documentation can be sparse, out of date, or simply wrong [2]. As a result, the set of configuration options can be large and incompletely documented [3], [4]. Human support may not be readily available. Developer time is a limited resource, and inspecting and diagnosing error messages displayed when users misconfigure software is a low priority.

Consider Hadoop, a widely used open source filesystem and MapReduce implementation. It is a well-established open source project, with dozens of active developers and ample documentation, including several books. Users range from large software companies with dedicated administration teams to hobbyists attempting to configure a small cluster at home. Despite this maturity, configuration error checking is still weak. If a user attempts to launch a recent version (0.20.2) of the Hadoop Filesystem with the default out-of-the-box configuration, it will crash with a null pointer exception, affording users little guidance about what to fix. We will use this as a running example.

This paper describes a technique that can help users troubleshoot this sort of startup error. When a user faces an unhelpful error message, such as Hadoop's "NullPointerException at line 134 of NetUtils.java", our approach points them to a configuration option that, if changed, will make the error go away. Our goals are to give users an answer as quickly as possible, without users installing any new debugging or analysis tools, without them having to understand the program's source code and without exposing sensitive site-specific configuration.

Yin *et al.* have studied configuration errors in five significant applications; they found that mistaken parameter values are 70-85% of all misconfigurations, and that invalid values, of the sort our technique could potentially address, are 40-50% of these [5]. Brown *et al.* suggest that typos are omnipresent and are a major problem [1]. Hence, looking for a specific "wrong" option will catch a large fraction of real-world misconfigurations.

Our chief tool is static analysis, which can be done independent of the user's query and whose results can be shared across users. Most prior work on configuration debugging has relied on large user communities or on modifying the program's execution environment. Both of these are deployment challenges. In contrast, our approach lets developers shield users from the complexity of diagnosis.

A. Our Contributions

The core of our approach is to analyze the program in question, producing a table mapping each line in the program's source code to the set of relevant configuration dependencies at that point. We envision this being done by the developers at release time. When a user encounters an error, they can use the error message to query this table, perhaps via a web service. Previous work has shown how to map log messages back to the origin line in the source code [6], [7]. Our approach requires no reconfiguration, new tools, or program modifications on the part of the user, unlike replay-based approaches or delta debugging. It requires no alterations to the JVM or standard library. This distinguishes our work from competing techniques such as dynamic taint tracking.

In managed environments such as the Java runtime, unhandled errors often result in a stack trace. We show that these stack traces can significantly improve the precision of analysis. Our technique, which we call *failure-context-sensitive analysis* (FCS), re-analyzes the call chain corresponding to the stack trace, pruning out irrelevant paths. By reusing the results from a prior static analysis, the run time for FCS can be kept

low. For Hadoop, the cost of these queries is approximately a minute with our current implementation. The results of these queries can of course be cached, reducing the time to answer for subsequent queries with the same stack trace. Our vision is that this analysis would be performed by a web service; users would need only to paste in a stack trace or log file to get back a diagnosis.

We have a prototype implementation targeting Java bytecode programs. We use the JChord analysis toolkit [8]. All the code used for the measurements in this paper is publicly available in the JChord source code repository[1]. We chose JChord because it was relatively easy for us to understand and modify its source code, and because its datalog-heavy programming style facilitated rapid prototyping. This was purely an implementation choice. Other dataflow or slicing engines should work comparably well. More broadly, we do not claim that our analysis algorithms are the best possible, rather, we show they are effective enough to be of use in solving an important practical problem, while still running quickly on large programs.

In previous work, we described how static analysis can find and categorize configuration options in many programs [4]. The analysis presented here is similar in spirit, and uses the techniques presented there for a preliminary part of the analysis. Explaining errors, however, requires a substantially more complex analysis than merely finding types. None of the techniques described here were present in our prior work.

B. Methodology and Organization

Our analysis is targeted to large complex software systems, such as Hadoop. In these systems, data will flow in and out of the system via the network and the filesystem. There may be native-language code. These data flows are difficult to capture dynamically, and even harder to model statically. As a result, we accept that our analysis will be imprecise and will miss some configuration dependencies. There will be both false positives and false negatives. We believe this is acceptable, so long as the analysis performs well in the common case, giving a correct diagnosis and not too many wrong guesses.

While our focus is on static analysis, we evaluate the benefits from several kinds of run-time instrumentation. This lets us gauge the sources of imprecision in our static approach. We show that tracking which options are read by the program can substantially improve analysis precision. This information can be recorded by the program and incorporated into the analysis cheaply. Only normal logging is required, not any sort of dynamic tracing or taint tracking.

The rest of this paper is organized as follows. We begin by describing our model for configuration options and give an overview of the analysis techniques we propose. In Section III, we present the details of our analyses. Our evaluation is in Section IV. Section V discusses limitations and sources of experimental error. Section VI describes related work. We give our conclusions in Section VII.

[1]http://code.google.com/p/jchord/source/browse/trunk/conf_analyzer

```
NetUtils.java:

SocketAddress getNameNodeAddress() {
60:   return createSAddr(
        getDefaultUri().getAuthority());
}
...
URI getDefaultUri() {
100:   return Conf.get("fs.default.name",
      "file:///"));
}
...
133: SocketAddress createSAddr(String t) {
134:     int colonIndex = t.indexOf(':');
135:     ....
```

Fig. 1. Simplified Hadoop code that produces null pointer exception with default configuration.

```
...
NetUtils.java 60 depends on fs.default.name
NetUtils.java 134 depends on fs.default.name
...
```

Fig. 2. Analysis output for code in Figure 1. Note that reading a variable, as on line 60, is not a use of the variable.

II. Model and Overview

We model configurations as a set of key-value pairs, where the keys are strings and the values have arbitrary type. This is a convenient abstraction for analysis as well as a close match for many standard configuration APIs. It is the abstraction offered by the POSIX system environment, the Java Properties API, and the Windows registry. It is also used by many applications for their own configuration data. It is not the only configuration model in use. Some systems use structured XML formats or other more complex models. At least sometimes, though, these more complex models can be approximated by treating an XPath to the option as its name. In previous work, we attained high accuracy using this XPath-based reasoning for Cassandra, which uses structured XML for configuration [4].

A. An Example

This section gives an example of how our analysis can diagnose a configuration error. We continue our running example: Hadoop, when started with default configuration, prints "NullPointerException at line 134 of NetUtils.java". Figure 1 is a simplified version of the code in question. The problem arises as follows. The `getNameNodeAddress` method attempts to construct an address from the authority (server) portion of the filesystem URI. The default filesystem URI (controlled by option `fs.default.name`) is `file:///`. This URI has no authority portion and so `URI.getAuthority()` returns null. This null then propagates to `createSAddr()`, which attempts to dereference it and then crashes. In the real implementation, this code is scattered across three different classes. Tracking down the problem in the source code would be a substantial task, particularly for new users.

Our approach builds a table mapping each line in the program to the options most directly associated with it. In this case, there will be an entry saying that `fs.default.name` affects line 134 of NetUtils.java. This table can then be presented to users via a web service (or a general-purpose search engine), letting them discover that `fs.default.name` is a relevant option at the point where the exception was raised. Our analysis does not tell users what value for the option will resolve the problem, but it can avoid wasted time tinkering with irrelevant options.

Before describing our analysis in detail, we give a sketch of how our analysis ties together the option and the relevant source code line. This is intended to give the overall flavor of the approach. The analysis marks the call to `Conf.get()` as an option read. Its return value is therefore assigned the label `fs.default.name`. Dataflow analysis tracks the flow of this label into `createSAddr()`. Line 134 of NetUtils.java uses this value, and so the analysis outputs that the line in question depends on `fs.default.name`. Figure 2 depicts this. Note that *reading* a configuration option, as happens on line 100, is not a *use* of the option.

B. Overview of Approach

The next section will discuss our analysis algorithms in detail. Here, we give an overview of the approach. We define a label for each configuration option. We then use dataflow analysis to determine which values and program points are associated with each label. This analysis happens at the bytecode level. At the end, we map these results back to line numbers. Since the analysis is being done at development time, we assume that debugging information is available to supply the line numbering. These error-attribution maps are small enough to easily fit in memory, even for large programs, as discussed in Section IV-C.

The steps in our analysis are listed below. The first step is a standard points-to analysis. The second step is shared with our previous work [4]. The next three steps are responsible for mapping program points to data dependencies. Last, we do a method-local control-flow analysis. Below is an outline of the approach:

1) Points-to analysis and call-graph construction.
2) Find configuration read points and associated names.
3) Create flow summaries for all reachable methods.
4) Dataflow analysis of configuration labels.
5) Optional: Demand-driven Failure-context-sensitive dataflow analysis of failure path using stack trace.
6) Method-local control-flow analysis using results of either whole-program or failure-context-sensitive dataflow analysis.

The basic analysis can all be computed statically and shared across all errors to be diagnosed. Only the (optional) failure-context-sensitive analysis needs to be repeated for each distinct error message.

Our approach is similar to taint tracking in that we are concerned with tracking the flow of labels through a program. Labels are introduced via configuration reads, and propagate via assignment and via library calls. Unlike taint tracking, we are not trying to find all possible dependencies, but only the most relevant ones for troubleshooting. To avoid the well-known problem of taint explosion [9], we apply a number of heuristics, discussed below. Our analysis can also be thought of as an application of thin slicing [10]. We are effectively computing a forward (thin) slice from each configuration option read point, and recording the set of slices that each program point belongs to.

Several design choices were forced on us by the programs we sought to analyze. We cannot assume that analysis will find an allocation site for every object. The programs we analyze are large, complex frameworks. Objects can be allocated in native code or in user code not present at analysis time. Hence, we apply labels to variables and to fields of objects, not to allocation sites. (This is similar to the approach taken in RacerX [11].) Second, we do not analyze inside the standard library. Instead, we treat library methods as opaque and treat references to library-defined types as primitives. These two aspects are complementary. Treating the library as opaque means that there will not be an allocation site associated with references returned by library methods. Hence, it makes sense to label the references themselves.

III. Implementation

This section describes our analysis in more detail, focusing on distinctive or unusual aspects. Readers seeking more information are referred to our implementation, which is publicly available. We first describe our core static analysis, followed by our failure-context-sensitive technique and the dynamic analyses we used to evaluate sources of imprecision.

A. Static Analysis

Points-To: We used the k-object-sensitive points-to analysis [12] built into JChord (with $k = 2$). The analysis is field-sensitive, path insensitive, and flow insensitive. It uses the static single assignment form of the program to gain some of the benefit of flow sensitivity (as suggested by Hasti and Horwitz [13]). We handle reflection using the technique presented in [14].

Many of the programs we analyzed make significant use of remote proceedure calls (RPCs). Labelled values should flow from the initialization code, invoked from main, to RPC methods, invoked remotely. This requires that the same abstract object be used in each context. To incorporate remotely-invoked methods into our points-to analysis, we automatically generate Java stubs that call each remotely accessible method of a server object. We then hand-wrote a few more lines of Java to ensure that these remote methods were invoked on the server object created by `main`.

Finding Configuration: We find configuration options using the approach described in our previous work [4]. Each of the applications we examined used a handful of "configuration" classes (often with that name) responsible for reading a configuration file and exposing a key-value interface to the rest

of the program. We manually construct a list of "configuration methods", including both API methods like `getenv` and their application-defined equivalents. For the programs in our study, this required a handful of per-application definitions. This list also records which argument corresponds to the option name. Configuration is typically spread across several domains: an environment variable named `someOption` may have no connection to a JVM property of the same name. To remove this ambiguity, we associate each configuration method with the name of the relevant domain.

At each call site that reaches one of these methods, a string analysis determines the name of the option being read, or "Unknown" if the analysis cannot find it. We used a custom-written string analysis that integrated easily with our points-to analysis. In prior work we showed that this technique finds options with over 95% accuracy on large complex programs, including the ones analyzed in this paper, making a more sophisticated string analysis unnecessary for us. More sophisticated string analyses (such as JSA [15]) could be used without changing our overall approach to finding configuration read points.

Dataflow: As noted above, we use an object-sensitive dataflow analysis. This decreased false positives by 40% compared to context-insensitive analysis. Beyond this, we found that our results were not extremely sensitive to the details of the analysis. We briefly summarize the implementation choices we made, but due to space limitations, we do not give detailed comparisons.

We mark a method's return value as depending on an argument if there is a control-dependency between the return statement and the argument. This rule is not strictly necessary; adding it increased coverage slightly (two additional true dependencies caught) while decreasing precision by approximately 10%.

If two variables may be aliased locally, we propagate labels from one to the other. Likewise, if a local variable aliases a static or instance field, and that variable becomes tainted via a library call, we taint the field. In the interests of precision and run-time performance, we do not model arbitrary interprocedural aliasing. One of the reasons this works is that most important Java library-defined types (including files and strings) are immutable. Even if an object is aliased, labels cannot flow between contexts; all the labels for an immutable object must appear in the context where the object is created. (This is not simply a happy accident. One reason why the Java designers preferred immutable types was because they are easier to reason about [16].)

Summarization: We use a summary-based context-sensitive analysis to gain additional precision. The approach is based on that described by Reps *et al.* [17]. When a label can flow from a method argument to its return value without going through the heap, we mark the method as having a functional dependence on the argument. Then, on the caller side, we use this summary to model the behavior of the function with respect to its arguments. These summaries handle dataflow paths that go purely through local variables. We also do a standard whole-program dataflow analysis, including the heap, covering all other paths.

The primary benefit of the summarization is that it simplifies our failure-context-sensitive analysis. It has two other benefits as well. It makes our dataflow more precise, leading to a 25% reduction in false positives for our test programs. It also makes the analysis faster: analysis times for the component programs of Hadoop went from an average of 10 minutes to an average of less than five.

Library Modeling: We adopt an approximate and pessimistic model for library code. We were driven to this model by the need to accommodate native code. We then found that it was practical to use for all library code, reducing the size of the code that needs to be analyzed and gaining the effect of an extra level of context-sensitivity for library calls.

We apply this model to the Java run-time libraries as well as to libraries used by the applications in our study. As a special case, we mark the Path types in Ant and Hadoop as library code. These types are returned by configuration-read methods and we need to treat them as opaque to maintain our invariant that labels apply only to primitive types and to references to library types.

We assume that if a parameter to an API call depends on a configuration option, then the object on which it is invoked and the return value both also depend on that option. As a special case, we do not mark input or output stream objects as depending on the data being written or read. We distinguish a handful of methods such as `equals`, where the arguments influence the return value but do not alter the object on which they are invoked.

Collection classes (such as arrays, hash tables, and so on) are mutable, and so our naive library model was inappropriate here. Our response was to use a simplified model implementation in Java. (Simply analyzing the library directly would not have captured the implicit dependency between the argument to `get` and its return value.)

Control-Dependence: We follow our dataflow analysis with a static and method-local control-flow analysis. Our primary concern was to capture the common programming idiom of checking a value and then emitting an error message if something is amiss. The analysis marks a program point as depending on an option if that point is on one side of a branch, where the branch condition has a data dependence on the option. Previous work on thin slicing has demonstrated that this sort of method-local analysis is likely to capture most relevant control dependencies while including few extraneous ones [10]. We have explored alternative slicing approaches that included additional implicit flows; these caused only small changes to our results.

The final output at each program point is the union of control and data dependencies at that point.

B. Failure-Context-Sensitive Analysis

Failure-context-sensitive analysis (FCS) is our technique for using stack traces to refine the precision of our error diagnosis. Our functional-dependence model (introduced in the previous

section) separates the effects of formal arguments and those of the heap or configuration read calls. As a result, we can re-analyze the failure path using only the labels for formal arguments propagated down the failing call path but using our full model for the heap. We compute the failure path by finding the program points that correspond to method calls in the stack trace supplied by the user. If a stack trace does not include a complete path from `main` to the failure point, we use the static analysis results to label the arguments to the lowest method on the stack trace.

Returning to our running example, the failing method, `createSocketAddr`, is called in many places in the code, with many different configuration options. Hence, knowing the options associated with the line where the exception is thrown is not very helpful. Even knowing the immediate caller is insufficient. The example code in Figure 1 above is slightly simplified. In fact, the failing function is generally called via a wrapper that specifies a default port number. Hence, several stack frames would be needed to disambiguate which option is responsible. (Different call paths require different levels of context to get a precise diagnosis.)

Unlike typical context-sensitive analysis algorithms, FCS works well on deep stack traces. This makes it well adapted to the common pattern of having several related utility methods that call one another with slightly different argument lists, such as with default values. In our experiments, this extra "deep" context made a major difference to results. Simply picking the calling context defined by the stack trace did not give comparable precision.

C. Dynamic Approaches

The static analysis above has several sources of imprecision. To gauge their importance, we replaced portions of our static analysis with instrumentation-based dynamic analysis.

The value of a configuration option that is never read cannot affect the program. Our first dynamic approach, *instrumented configuration reads*, records which options are read and where. This is then consumed by the static analysis discussed above. Effectively, this is static analysis using only the options actually read by the program.

Our second dynamic approach, *instrumented configuration flow*, goes farther. In addition to dynamically monitoring option reads, we track the flow of labelled objects through the program dynamically. When a configuration value is returned by a method, that value and the associated option name are recorded in a lookup table. This table is kept in the memory of the instrumented process. At each new invocation, the options associated with each parameter are written to disk. Unlike our static analysis, this approach tracks objects, not values. We augment the dynamic tracking with a method-local static analysis to track primitive types and the flow of values that were null at run-time. We reuse our static control-dependence analysis.

Instrumented configuration flow was intended purely to measure sources of imprecision. In contrast, instrumented configuration reads could be incorporated into existing systems.

TABLE I
RESULTS USING CONFERR FOR FAULT INJECTION: COVERAGE IS HIGH AND FALSE POSITIVES ARE LOW.

Target	Errors Injected	Avg False Positives	False Neg.	Success rate
Hadoop HDFS	5	1.0	1	80 %
Hadoop MapRed	6	3.5	1	83 %
JChord	7	2.1	1	85 %

In the programs we have seen, configuration data is accessed exclusively via a handful of program classes. The value of each option could be logged the first time it is read. The volume of information would be small and proportional to the number of configuration options. This information could be extracted from log files during troubleshooting and used to filter the static analysis results. This logging would also compensate for imprecision in finding option names by making the concrete run-time names available.

IV. EVALUATION

In this section, we evaluate the static analysis approaches we presented above. We seek to answer several research questions: How complete and correct are the results of the analysis? What are the sources of imprecision? How much gain is there from failure-context-sensitive analysis?

We performed two different sets of experiments. First, we performed a fault injection study using two programs, Hadoop and JChord. This experiment measures how well our static analysis works, how much gain there is from failure-context-sensitive analysis, and how much different sorts of dynamically-collected information improve precision. Our second set of experiments controls for selection bias in the injected faults and the programs we investigated. We measured the statistical properties of our analysis on a range of additional programs and program points. We demonstrate that the programs and program points evaluated in the first set are not anomalous, evidence that our technique is more broadly applicable.

A. Catching Injected Configuration Errors

Our fault injection experiments focused on two software systems, Hadoop and JChord. The Hadoop source code consists of several hundred thousand lines of Java, and over 20 library dependencies. Its functionality is split across several separate but communicating programs. It makes heavy use of reflection, making it a suitably "hard target" for program analysis. JChord is a program analysis tool and deadlock detector. It has five developers, approximately 30,000 lines of Java source code and a dozen library dependencies. It too makes heavy use of reflection. Both programs support several dozen options.

We used the ConfErr tool [18] to insert typographic errors into otherwise-working configurations for each of JChord, Hadoop MapReduce, and Hadoop's HDFS filesystem. Some typographic errors are masked silently by the program. Each of the remaining erroneous configurations led to either an error

message or a stack trace. If there was an error message, we check for dependencies at the line where the message was printed. For stack traces, we use the first point on the trace with a dependence. (This rule covers cases where an exception is raised inside unanalyzed library code.) We analyze the results with our basic static analysis.

Our results are displayed in Table I. ConfErr found 18 errors across the three systems under test. Of these, our tool diagnosed all but three, a 17% false negative rate. (The false negative rate is the fraction of errors for which our algorithm did not find the true injected root cause.) Once each for Hadoop MapReduce and HDFS, an injected string broke the XML format of the configuration file. This caused an error at configuration read time. Our tool was unable to diagnose these errors, because they were not tied to any particular option. The un-diagnosed JChord error was caused because the bad value was used to set up the command line for a child process; our analysis does not capture dependencies between command line arguments and configuration options. We compute the false positive rate by counting the number of configuration dependencies other than the one with the injected error. This averaged between 1 and 3.5 for the systems in question. (The ideal is 0, meaning exactly one diagnosis for every error.) Our technique thus succeeds in drawing attention to a handful of possible problem diagnoses.

Next, we examine precision more closely by comparing five different analysis techniques: static analysis, the two dynamic approaches discussed above, failure-context-sensitive static analysis, and failure-context-sensitive analysis using only options read at runtime. For errors without stack traces, FCS is equivalent to static analysis. We mark these cases with an X in our data tables and reuse the result of the static analysis. All our techniques had the same false negative rates, so average false positives is the relevant figure of merit.

We felt that the errors found by automated fault injection were not necessarily representative of the full range of user mistakes. Errors such as an unavailable port number for a server or insufficient disk space show up in operation but not in automated testing. For Hadoop, we found a number of mailing list messages with errors, configuration, and a confirmed diagnosis. We incorporated those into our test inputs to provide a wider range of test cases. They are marked with asterisks in Table II. For JChord, we reused the errors found by automated testing. Our results are presented in Tables II for Hadoop and III for JChord.

For both Hadoop and JChord, the biggest precision gain came from dynamically recording which options are read (the *instrumented reads* approach). Dynamically tracking values (*instrumented flow*) adds additional benefit. The gap between instrumented reads and instrumented flow is a measure of how much imprecision comes from our dataflow and points-to analyses. As can be seen, there is a gap, but a comparatively smaller one.

For Hadoop, in the cases where static analysis finds many possible options, failure-context-sensitivity improves precision substantially, reducing the average number of guesses by approximately a third. Failure-context-sensitivity plus restricting to options read at runtime reduced false positives by nearly a factor of four, as compared to pure static analysis. For JChord, FCS was ineffective; JChord errors largely lacked stack traces, and so the technique does not apply.

A few aspects of our measurements require explanation. Hadoop Test 3 is a real user-reported error that manifests in a subprocess spawned by the Hadoop TaskTracker process. As mentioned, our analysis does not track this type of dependency. Instrumentation-based approaches can sometimes find more options than were found statically. Our static analysis combines related option names. At run-time, `fs.hdfs.impl` and `fs.file.impl` are distinct, but our static analysis treats the two as one option, `fs..*.impl`. Tracking objects can also introduce spurious option dependencies not found statically; we observed this in our fourth JChord test case.

We also tried a simple heuristic, *last-option-read*. This heuristic looks at the last configuration option that a program read before failing, but does not examine the program structure in any way. We had expected this to work because many wrongly-set options will result in errors on first use, if they will result in errors at any point. Experimentally, we see that this heuristic works half the time for Hadoop: Hadoop often reads options immediately before using them; if the first use of an option value triggers an exception, then that option was likely the most recent one read. The heuristic does not work at all for JChord. JChord reads most of its options at program startup, so there is no close connection between reads and uses. We conclude that this heuristic relies entirely on particular styles of programming for its effectiveness.

B. Measurements From Other Programs

In the previous section, we only looked at configuration errors in two software systems, Hadoop and JChord. To estimate how well our approach would work on other errors and other programs, we compare aggregate statistics from these two programs to those from several others.

The programs we analyze span a range of applications and have a wide variety of developers. Apache Ant is a replacement for `Make`, originally developed at Sun Microsystems as a component of the Tomcat servlet container[2]. HBase is a re-implementation of Google's BigTable storage architecture developed by a loose collection of open-source developers spread across several companies[3]. Cassandra is a distributed storage service developed at Facebook [19]. FreePastry is a peer-to-peer distributed hash table originally developed at Rice [20]. Hence, these programs represent a range of programmer expertise and style.

To estimate precision, we look at the average number of exceptions at method call points. (Non-method call statements cannot print error messages and can only raise exceptions in a few circumstances.) We display our results in Table IV. We separately display the average number of dependencies at points with at least one.

[2]http://ant.apache.org/

[3]http://hbase.apache.org/

TABLE II
DETAILED RESULTS FOR HADOOP. SHOWS COUNT OF FALSE POSITIVES. "N" = TECHNIQUE FAILED. "X" = INAPPLICABLE (NO STACK TRACE). AVERAGE FALSE POSITIVES FOR EACH TECHNIQUE IS KEY FIGURE OF MERIT.

Run ID	Program	Error	Static	Dynamically-measured		FCS		Last
				Reads	Flow		+ Dyn. Reads	Load
1	TaskTracker	rpc socket factory class not a class	0	0	0	X 0	X 0	Y
2	TaskTracker	master hostname not set	7	0	0	6	0	Y
* 3	TaskTracker	child work dir not writable	N 0	N 0	N 0	N 0	N 0	N
* 4	NameNode	storage dir not writable	3	1	0	3	1	N
5	NameNode	fs.default.name not this machine	0	0	0	0	0	N
* 6	NameNode	fs.default.name not HDFS	5	0	0	0	0	Y
7	NameNode	Data port unavailable	1	1	0	1	1	N
8	NameNode	Info. port in use	2	2	0	2	2	N
9	NameNode	invalid topology mapping class	0	0	0	0	0	Y
10	NameNode	topology mapping script not valid	4	4	1	4	3	N
11	NameNode	FS object quota exceeded	0	0	0	0	0	N
12	NameNode	malformed XML	N 0	N 0	N 0	X 0	X 0	N
13	JobTracker	jobtracker info port in use	2	2	0	2	2	N
14	JobTracker	storage dir not writable	8	3	1	8	2	N
* 15	JobTracker	master hostname not set	1	0	0	0	0	Y
16	JobTracker	log dir not set	0	0	0	X 0	X 0	Y
* 17	JobTracker	carriage return at end of address	1	0	0	0	0	Y
18	JobTracker	malformed XML	N 0	N 0	N 0	X 0	X 0	N
19	DataNode	missing DN port number	8	3	1	7	2	Y
20	DataNode	use of deprecated option	0	0	0	X 0	X 0	Y
* 21	DataNode	master host not specified	8	0	0	0	0	Y
22	DataNode	storage dir not writable	0	0	0	X 0	X 0	Y
Success %			86.4	86.4	86.4	86.4	86.4	50
Average False Pos			**2.3**	**0.7**	**0.1**	**1.5**	**0.6**	

TABLE III
DETAILED RESULTS FOR JCHORD.

Run ID	Program	Error	Static	Dynamically-measured		FCS		Last
				Reads	Flow		+ Dyn. Reads	Load
1	JChord	no main class	1	1	0	1	1	N
2	JChord	no main method	0	0	0	X 0	X 0	N
3	JChord	no such analysis	3	0	0	X 3	X 0	N
4	JChord	invalid context-sensitive analysis name	1	1	2	1	1	N
5	JChord	printing nonexistent relation	0	0	0	0	0	N
6	JChord	disassembling nonexistent class	0	0	0	X 0	X 0	N
7	JChord	invalid scope kind	4	2	0	X 4	X 2	N
8	JChord	invalid reflection kind	4	2	2	X 4	X 2	N
9	JChord	wrong classpath	N 2	N 2	N 1	X 2	X 2	N
Success %			88.9	88.9	88.9	88.9	88.9	0
Average False Pos			**1.7**	**0.9**	**0.6**	**1.7**	**0.9**	

Most program points have no configuration dependency. Of points with a dependency, the average number at any point is less than eight for all the programs in our sample, and less than six for all but two programs. As can be seen, Hadoop and JChord (the programs examined in detail above) are not radically different from the other programs we examine here. This suggests that our failure injection results are generalizable to other programs. Moreover, the average number of dependencies per program point found here is similar to the average number of false positives measured in the failure injection experiments.

This further validates our methodology. It shows that the program points at which errors arose in the experiments above are comparable to the average for each program as a whole. (Note the average number of false positives is lower than the average number of dependencies: one option is actually responsible for a given error in our tests.)

In some programs, log messages explicitly mention configuration options. Programmers may print the value of an option alongside its name or produce error messages with option names. These messages are effectively labelled data, and give us an additional avenue for estimating the false negative rate for our analysis: find messages that mention options, and check whether the analysis finds a dependency between that message and the named option. The results of this analysis are also displayed in Table IV. All but a handful of these "explicit dependencies" are detected by our technique. We saw two causes for false negatives. First, our analysis does not track control flow via exceptions. Second, some messages mention an option that does not actually influence the message, such

as "option X is deprecated, use Y instead." This latter case is a limitation of our evaluation technique, not of the underlying program analysis.

C. Performance Aspects

We next consider the running time and output size of our analysis. Time costs for all the programs we studied are displayed in Table IV. Measurements were conducted on a modern dual-core laptop with 4 GB of RAM. The largest program and longest-running analysis was Ant, which took just under half an hour. Most others were under 10 minutes. This is acceptable as part of a nightly build process.

Our prototype outputs flat, uncompressed, text, with one line for each (line,option) dependency pair. (Table 2 gives an example of this.) Each entry is approximately 150 bytes of uncompressed ASCII. Even with this space-inefficient format, the largest table we have seen, for Ant, is 3MB. This can be kept in memory and searched very quickly, taking less than a second. A more space-efficient output format could reduce space and time cost substantially.

Unlike our base static analysis, failure-context-sensitive analysis has a measurable run-time per error. For Hadoop, the average FCS run time was approximately a minute; or roughly a quarter of the static analysis time. For JChord, FCS took approximately 30 seconds, again a quarter of static analysis time. This shows that most of the analysis can be shared across runs. While we did not optimize aggressively for performance, it is already good enough for some interactive uses.

V. Discussion

This section discusses the limitations of our work. We start with experimental flaws and broaden our scope to limitations of the technique and the underlying model.

There are programming practices that would reduce the effectiveness of our analysis technique. In particular, our techniques would produce useless results on programs that catch all exceptions and emit only a general error message. Our log-based experiments (though not our technique) implicitly assume that messages that mention options are representative of program points that users care about.

Our technique focuses exclusively on configuration errors where the value of an option is wrong and this causes a program to fail in a deterministic way with an error message. This category represents many but not all, real misconfigurations [5], [1]. The injected errors we used to evaluate our analysis are intended as representative examples.

We focus exclusively on named configuration options with a key-value semantic. This model is common, but not universal. It has two properties that we rely on. The first is that named options are associated nearly one-to-one with points where the program reads the option, and this mapping can be found automatically. The second is that options are not arbitrary data. In our previous work we showed that a large majority of options fall into one of three categories: numerical parameters (such as timers and buffer sizes), named modes of operation (such as Boolean switches that enable and disable features), and names of system-level entities (such as network addresses and file names). Values of these types are used in narrower ways than arbitrarily chosen program inputs would be, including those inputs that might be viewed as configuration under a broader definition. For example, our approach would not help pinpoint which line of a malformed program caused a compiler to abort.

Our prototype and experiments were restricted to Java. The techniques we outline might not work as well for languages such as C where the weaker type system can make points-to analysis more challenging. As with all static analysis approaches, our techniques would not work to diagnose problems in "programs" consisting of multiple components in several languages, such as problems caused by a program's start script. And as shown above, our analysis does not track options that are passed between processes via the command line. We tried to pick complex and highly reflective programs, but more complex and harder-to-analyze programs certainly exist.

We do not address errors that manifest themselves as performance problems, resource exhaustion, or silent failures. Addressing these errors will require a revision to our problem statement, not just algorithmic refinements. In these cases, there is no unique option or small set of options responsible for the problem. If a program runs out of memory, any configuration option that controls memory allocation could in principle be tuned to make the error go away — potentially a very large set. The options that control the preponderance of memory allocations will be workload-dependent. Therefore, dynamic analysis may be better suited to this problem.

We opted for dataflow analysis, rather than a more sophisticated symbolic analysis. Dataflow analysis will cover all reachable code and runs quickly. A more sophisticated analysis would need some model for how the Java runtime uses its parameters, which our technique does not require.

Our technique, on its own, does not tell a user *why* a value is wrong. In previous work, we proposed "configuration spellcheck" in which analysis extracts a type signature for an option such as "writable file". The two approaches could be combined. Given that an option is likely responsible for an error, the fact that the option's value should have been a writable path but was not is a good root-cause explanation. Generating concise descriptions of the problem with a given option is left as future work, as is ranking possible diagnoses.

VI. Related Work

We now summarize the prior work on diagnosing configuration problems in computer systems. We group these techniques into four broad areas: program-analysis approaches, signature-based approaches, inter-host comparisons, and replay techniques. (There has also been prior work in analyzing compile-time configuration; for example, Krone and Snelting discuss finding flawed configuration models [21]. We lack space to further discuss this area.)

We are aware of two instances of prior work using program analysis for configuration debugging. Sherlog takes as input a program and a log, and uses a SAT solver to infer the state of the program prior to failure [6]. Like our work, this is a

TABLE IV
MEASUREMENTS SHOWING ANALYSIS COVERAGE. TABLE SHOWS NUMBER OF METHOD CALL SITES OBSERVED BY ANALYSIS, TOTAL OPTIONS FOR PROGRAM, OPTIONS PER CALL SITE, AND OPTIONS PER CALL SITE AT SITES WITH AT LEAST ONE.

Program	Analysis time (min:sec)	options	method calls	calls with dependence	avg deps per call	avg deps/call if > 0	opt mentions in logs	detected mentions
Ant	28:08	73	34071	6579	1.4	7.2	14	11
Cassandra	3:02	50	3693	667	0.7	4.1	0	0
FreePastry	23:50	119	24193	6221	1.3	4.9	0	0
HBase	7:15	140	15122	3815	1.3	5.2	3	3
Hadoop DataNode	3:35	72	7258	1766	0.7	2.9	12	11
Hadoop JobTracker	4:04	126	8939	2281	1	3.9	9	7
Hadoop NameNode	4:20	112	11739	3161	0.7	2.7	5	5
Hadoop TaskTracker	4:58	113	8112	3170	2.3	5.8	12	12
JChord	2:20	90	6873	2234	0.8	2.4	63	61

static approach that does not require modifying the execution environment. Unlike our work, the analysis requires output from the faulty program. Users might need to wait as much as half an hour for an answer. In our approach, users can get an immediate answer from the pre-computed static analysis. Our failure-context-sensitive analysis takes no more than a minute or two.

ConfAid uses dynamic taint tracking (and short-distance speculative execution) to explain errors [22]. ConfAid tracks tokens from specified "configuration sources", and is able to pinpoint the tokens that most directly lead to an error. This technique is likely to be more precise than ours and encompasses a broader definition of configuration. There are two disadvantages in comparison to our work. ConfAid requires the user to modify the execution environment for the program being diagnosed. Our approach uses only the generated error message. Our approach can attribute errors caused by an inappropriate default value for an option, while ConfAid can only track options that are explicitly set.

Problem-signature approaches diagnose configuration errors by extracting a signature of the program behavior associated with a particular misconfiguration. This requires that a library of problem signatures be created for each program. A program's pattern of system calls is commonly used for this. Unfortunately, signature collection (needed both from faulty and correct runs of a system) is expensive. Further, signatures cannot always be shared across sites, since they include the specific configuration on each machine [23], [24].

Instead of examining program behavior to diagnose errors, it is also possible to compare configurations themselves. Strider and PeerPressure can help identify problems caused by bad Registry entries on Windows machines. Strider [25] tries to pair a working and non-working machine and constructs the set of configuration differences. It then takes the intersection of this set with the set of Registry entries read while the user attempts to perform the failing action. PeerPressure goes a step further, using the relative frequency of various settings as a clue to their likelihood of causing the problems [26]. Since most machines, it is assumed, are healthy, an unusual configuration setting read by a faulty process is likely the culprit. Similar approaches can be used to identify configuration problems in grid deployments [27]. All these techniques are limited by the need for large installed bases with relaxed privacy policies.

Replay-based diagnosis techniques automatically try possible configuration changes in a sandbox, allowing many diagnoses to be tested without the risk of overwriting correct configuration or damaging the rest of the system. Chronus, AutoBash, and Triage are all systems of this type [28], [29], [30]. Given a once-working system, Chronus uses virtualization and a customized filesystem to pinpoint the particular configuration change that caused the system to stop working. The AutoBash system uses kernel-level speculative execution to test the results of various configuration changes. Given a set of predicates (programs that return "passing" or "failing"), AutoBash can try potential configuration fixes in the background, without interfering with the rest of the system, until some set of fixes solves the problem. Triage takes a similar approach, speculatively replaying the events leading up to a failure with small variations to pin down the root cause.

Our program analysis approaches are complementary to replay. Program analysis makes replay more practical for niche or tailored software by producing a list of possibly-wrong options. Replay reduces the cost of imprecise analysis by automatically trying multiple possibilities.

Related to replay, delta debugging is an algorithm for pinpointing error causes, given a large set of potential changes [31]. Delta debugging relies on having a working state, a broken state, and a set of potential changes. This means it cannot diagnose errors where no working configuration is available for a particular site. It also pushes more of the work of troubleshooting onto the user site — potentially a serious bar to deployment.

VII. CONCLUSIONS

A number of conclusions emerge from our work.

Computing and storing configuration dependencies is tractable. In theory, each program point could depend on an arbitrary subset of the program's configuration options, resulting in excessive computation and storage costs. In practice, we have observed that most program points depend on only a small number of options. As a result, storing the dependencies

at each point is feasible, with size roughly proportional to the program's code.

Dataflow analysis can explain typo-induced errors. The high accuracy of our analysis implies that, at least for the programs we studied, there are usually data dependencies between configuration values and the points in the code where a bad value can cause an error.

For diagnosing configuration errors, it is safe to model library code, rather than analyzing it. Our approach uses an approximate model for library code, instead of analyzing the library and tracking the flow of options through it. This gave us substantial performance benefits without causing any false negatives in our tests.

Failure-context-sensitive analysis helps exploit stack traces for configuration debugging. In our testing for Hadoop, failure-context-sensitive analysis reduced the imprecision of static analysis by a third while taking approximately a minute per stack trace.

Programs should log the configuration options they read. If a program does not read all its options, then knowing which options were actually read substantially improves analysis precision. Recording this information only requires developers to insert a handful of log statements, rather than make extensive changes to their programs. Since even large systems (like Hadoop) often have only a few hundred options, the logging overhead is small.

Acknowledgements

We thank Koushik Sen and Mayur Naik for their advice and encouragement. We appreciate the time and attention devoted by the anonymous reviewers.

This research is supported in part by gifts from Google, SAP, Amazon Web Services, Cloudera, Ericsson, Huawei, IBM, Intel, Mark Logic, Microsoft, NEC Labs, Network Appliance, Oracle, Splunk and VMWare and by DARPA (contract #FA8650-11-C-7136).

References

[1] A. B. Brown and D. A. Patterson, "To err is human," in *Proceedings of the First Workshop on evaluating and architecting system dependability (EASY'01)*, 2001.

[2] M. Levesque, "Fundamental issues with open source software development," *First Monday:*, vol. Special Issue #2: Open Source, October 2005.

[3] M. Michlmayr, F. Hunt, and D. Probert, "Quality practices and problems in free software projects," in *First International Conference on Open Source Systems*, 2005.

[4] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *ICSE*, 2011.

[5] Z. Yin, J. Zheng, X. Ma, Y. Zhou, S. Pasupath, and L. Bairavasundaram, "An empirical study on configuration errors in commercial and open source systems," in *SOSP*, 2011.

[6] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," in *ASPLOS*, 2010.

[7] W. Xu, L. Huang, M. Jordan, D. Patterson, and A. Fox, "Detecting Large-Scale System Problems by Mining Console Logs," in *SOSP*, 2009.

[8] M. Naik, "JChord," http://jchord.googlecode.com.

[9] E. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *IEEE Symposium on Security and Privacy*, 2010.

[10] M. Sridharan, S. Fink, and R. Bodik, "Thin slicing," in *PLDI*, 2007.

[11] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," in *SOSP*, 2003.

[12] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, pp. 1–41, January 2005.

[13] R. Hasti and S. Horwitz, "Using static single assignment form to improve flow-insensitive pointer analysis," in *PLDI*, 1998.

[14] B. Livshits, J. Whaley, and M. Lam, "Reflection analysis for Java," in *Third Asian Symposium on Programming Languages and Systems*, 2005.

[15] A. Christensen, A. Møller, and M. Schwartzbach, "Precise Analysis of String Expressions," in *Symposium on Static Analysis*, 2003.

[16] J. Bloch, "How to design a good api and why it matters," in *OOPSLA*, 2006, pp. 506–507. [Online]. Available: http://doi.acm.org/10.1145/1176617.1176622

[17] T. Reps, M. Sagiv, and S. Horwitz, "Precise interprocedural dataflow analysis via graph reachability," in *POPL '95*, 1995.

[18] L. Keller, P. Upadhyaya, and G. Candea, "ConfErr: A tool for assessing resilience to human configuration errors," in *DSN*, 2008.

[19] A. Lakshman, P. Malik, and K. Ranganathan, "Cassandra: A Structured Storage System on a P2P Network," in *SIGMOD*, 2008.

[20] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware*, 2001.

[21] M. Krone and G. Snelting, "On the inference of configuration structures from source code," in *ICSE*, 1994.

[22] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *OSDI*, 2010.

[23] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma, "Automated known problem diagnosis with event traces," in *EuroSys*, 2006.

[24] X. Ding, H. Huang, Y. Ruan, A. Shaikh, and X. Zhang, "Automatic software fault diagnosis by exploiting application signatures," in *LISA*, 2008.

[25] Y. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. Wang, C. Yuan, and Z. Zhang, "STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support," in *LISA*, 2003.

[26] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic Misconfiguration Troubleshooting with PeerPressure," in *OSDI*, 2004.

[27] N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff, "Mining for misconfigured machines in grid systems," in *International Conference on Knowledge Discovery and Data Mining*, 2006.

[28] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration debugging as search: Finding the needle in the haystack," in *OSDI*, 2004.

[29] Y.-Y. Su, M. Attariyan, and J. Flinn, "Autobash: improving configuration management with operating system causality analysis," in *SOSP*, 2007.

[30] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, "Triage: diagnosing production run failures at the user's site," in *SOSP*, 2007.

[31] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *ESEC/FSE*, 1999.