AutoLog: Facing Log Redundancy and Insufficiency

Cheng Zhang[1], Zhenyu Guo[2], Ming Wu[2], Longwen Lu[1], Yu Fan[1]
Jianjun Zhao[1], Zheng Zhang[2]
[1]Shanghai Jiao Tong University, [2]Microsoft Research Asia
{cheng.zhang.stap, nowen, yuf, zhao-jj}@sjtu.edu.cn
{Zhenyu.Guo, miw, Zheng.Zhang}@microsoft.com

ABSTRACT

Logs are valuable for failure diagnosis and software debugging in practice. However, due to the ad-hoc style of inserting logging statements, the quality of logs can hardly be guaranteed. In case of a system failure, the log file may contain a large number of irrelevant logs, while crucial clues to the root cause may still be missing.

In this paper, we present an automated approach to log improvement based on the combination of information from program source code and textual logs. It selects the most relevant ones from an ocean of logs to help developers focus and reason along the causality chain, and generates additional informative logs to help developers discover the root causes of failures. We have conducted a preliminary case study using an implementation prototype to demonstrate the usefulness of our approach.

1. INTRODUCTION

Making complex software is hard, and getting them correct is often harder. Despite many progresses made by the research community, the adoption of advanced formal procedures and tools has been slow. Logging by *printf* is the predominant debugging practise, and will likely remain so for many years to come. In many ways, logging statements (i.e., the statements to print logs) can be collectively regarded as a "program" over the program under debugging. They represent the developers' effort to observe the inner working of the program, revealing expected or erroneous behavior. They are convenient to add, and do not create side effects, other than some (hopefully) minor runtime overhead.

This freedom comes with a cost. The practise of logging is ad-hoc. Logging statements accumulate over time by different developers, and the *quality* of the logs has never been a focus of either the research or the development community. Once hitting a bug, we are overwhelmed by the voluminous logs. Filtering by "grepping" reduces the overload to certain degree, but using the logs to reason about the causality is still difficult, since the relationships between the logs are obscure. Yet, many critical logs that could have led to quick discovery of the bug can still be missing.

Rooting out the bugs is ultimately the responsibility of the developer. Acknowledging the fact that logging is the critical debugging exercise, and that debugging is typically interactive, AutoLog attempts to provide a set of practical techniques to improve the utility of logs. More specifically, we make the following contributions:

1. *log slicing* combines program analysis techniques and information provided by the logs, and presents only the relevant (though conservative) logs, framed over causality chain.
2. if and when the developer requires further probing, *log refinement* inserts new logging statements, prioritized by their degree of uncertainty reduction.

This paper describes the prototype architecture of AutoLog. Our early experience of using it against a real-world complex software (Apache Hadoop Common) has shown the promise of the direction.

The rest of this paper is organized as follows. Section 2 gives an overview of our approach. Section 3 describes the technical details. Section 4 briefs the prototype implementation and shows the result of a case study. We discuss related work and our future plan in Section 5.

2. OVERVIEW

Target scenario. We imagine that AutoLog will be used in an interactive debugging session. Our target scenario is pre-release in-house development, where the source code is available. When the session starts, there is at least one log available, namely the one corresponds to the failure site. That starting log, which can be an error message, an assertion failure, or an exception, may or may not contain other rich information. For our prototype we have assumed that the stack trace and the variable that triggered the failure are available when the failure is reproduced for the first time. This is a constraint that we might remove in the future.

From that starting log, typically developers will explore backward, searching for an answer to explain why the bug has occurred; they might also explore along the execution flow forward to consolidate their understanding of the program behavior. When they cannot clearly pinpoint the root cause yet but have formed some clues, they may choose to insert additional logging statements in subsequent runs. Once they have identified the bug and installed the corresponding fix, the program is run again, and this cycle might repeat.

Debugging using AutoLog. Figure 1 shows the AutoLog architecture as well as the flow; our current prototype only

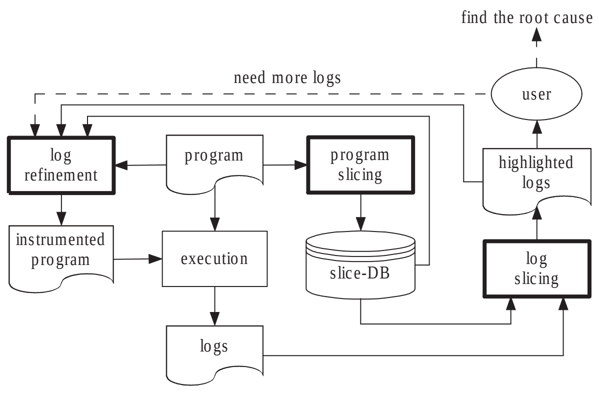

Figure 1: The Architecture of AutoLog.

contains part of the system. The components of AutoLog are called into play in different stages of the debugging session.

In the offline part of the process, AutoLog takes the source code and generates a conservative set of slices and stores them into a database (i.e., the slice-DB in the figure). A slice is the set of statements that may affect the value of a specific variable at a specific program location (the pair of variable and location is called *slicing criterion*). Slice-DB may contain only a subset of all possible program locations, as computing slice is expensive. Selecting good candidates for the slice-DB is itself an interesting topic. For now, we start with the logging statements, as they are inductive: the fact that the developer has added an entry there means that she wants to know more about the "what" at that point, and we might as well be prepare to answer the "why".

When the debugging session starts, AutoLog takes the starting log, and explores backwards from the closest entry in the slice-DB. During this step, it also scans through the logs, taking the runtime information that they might contain, and prunes the conservative slice(s) with an approach that we call *log slicing* (see Section 3). The end of this scan process will produce the set of logs that are aligned along the causality chain. Only these logs are presented to the developer. It is critical to optimize the performance of log slicing, as it directly affects the usability of the tool.

Typically, when the developer studied the logs and the relevant part of the code, she would have formed at least some clues as why the bug has appeared. If the exact root cause remains elusive, it means that more information is needed. New logging statements can be manually added, as is done today. AutoLog goes further, it computes and instruments new logging statements based on a set of heuristics to reduce ambiguity aggressively. This part of the process is called *log refinement*, and will be detailed in Section 3.

We believe that the above interactive debugging process is typical, in which AutoLog embeds naturally. This approach has a number of challenges, however. For instance, we assume that the bug is deterministic and can be triggered again. There is also the weakness associated with the slicing technique itself, such as the difficulty to handle aliases and shared variables across threads. We will delay the discussion on the challenges and limitations until Section 5.

3. APPROACH

The goal of AutoLog is to aid, not to replace, a programmer in her pursuit to discover and fix the bug. Since debugging involves reasoning about causality, AutoLog relies on slicing [5] as the underlying technology. The original program slicing is based on static (data and control) dependencies, thus the slice contains all the statements that **may** affect the slicing criterion in any possible execution. In contrast, dynamic slicing focuses on dependencies that occur in a specific execution. Therefore, the dynamic slice contains the statements that have **actually** affected the slicing criterion in one execution.

In the example shown in Figure 2, using statement 13 and variable `a` as slicing criterion, we get a static slice which consists of all the statements except statement 10. Statement 10 calls the logger which reads variable `a`; it could be included in the slice by data dependency if it wrote variable `a`. In an execution, if `p` is 1 and method `read()` returns 2 and 0 at statements 1 and 5, respectively, then the assertion at line 13 fails. In this case, the dynamic slice contains statements 2, 3, 4, 5, 6, 8, and 13.

```
    void m(int p){
1.  int a = read();
2.  if(p % 2 == 1){
3.    a = p;
4.    if(a > 0){
5.      int tmp = read();
6.      if(tmp > 0){
7.        a = a * tmp;
        }else{
8.        a = a * (-1) * tmp;
        }
      }else{
9.      a = a + p;
      }
10.   log.info(a);
    }else{
11.   if(a == p){
12.     a = 2;
      }
    }
13. assert(a != 0);
    }
```

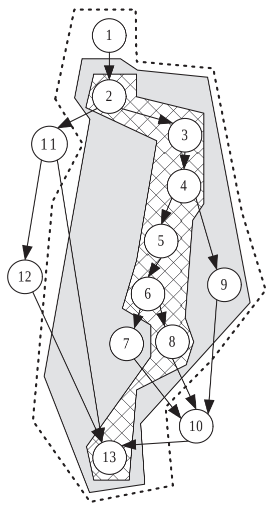

Figure 2: A program and its control-flow graph. Nodes surrounded by dashed border represent the static slice pruned by control flow relations. Nodes with pattern/shaded background represent the dynamic/hybrid slice. Note that the data dependency graph, which is necessary for computing the slices, is omitted for brevity.

Both static and dynamic slices rooted from the failure site (line 13) contain the culprit (line 5). However, the dynamic slice is far more precise. The tradeoff is that, while static slice can be computed offline, dynamic slice requires heavy instrumentation to acquire the entire execution trace, in addition to computation of the slice.

The core idea of AutoLog is to refine the scope of static slices using the dynamic information already made available from the printed logs. The advantage is that the expensive operations, such as computing dependency graphs and static slices, are strictly offline from a debugging perspective. Furthermore, when new logging statements are inserted, static slices do not change and can be reused. The online part of the refinement is hopefully very light weight, therefore maximizing the usability of the tool. To achieve this goal, log slicing in AutoLog is *hybrid*, in the sense that both static and dynamic information are used.

Given a log file, AutoLog first parses the textual logs and maps them to the corresponding logging statements. Cur-

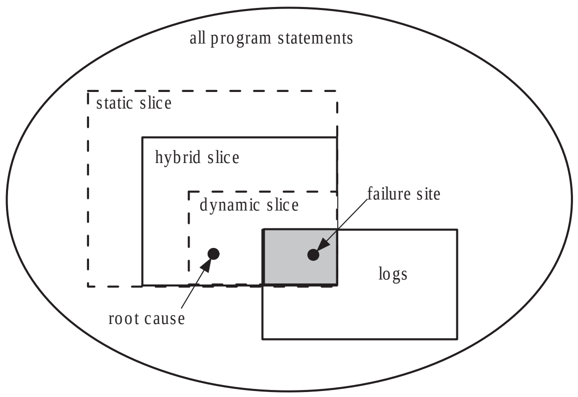

Figure 3: The relationship between different kinds of slices and logs.

rently we assume that the logs are produced with general logging frameworks, such as log4j, so that the exact locations of logging statements can be included in logs and easily parsed from logs. With this assumption, the mapping between logs and logging statements is straightforward. However, as discussed in prior work [6, 7], a more general log parsing technique is often necessary, since many systems actually have their own logging mechanisms. We are developing a log parser targeting object-oriented programs. It is an extension to the log parser proposed in [6] and will address the complexity of string manipulation based on [2].

The goal of AutoLog's log slicing is to classify statements in the corresponding static slice into groups of **must**, **may**, and **must not** and then refine the slice. This is done in two steps: 1) pruning and 2) re-slicing. The pruning step filters out from the static slice the statements that must not have been executed. This step is trivial to compute, an essentially reachability analysis starting from the executed logging statements. In Figure 2, the log printed by node 10 will prune away nodes 11 and 12. We can also conclude that nodes 1, 2, 3, 4, and 13 must have been executed, because nodes 1, 2, 3, and 4 dominate node 10 and node 13 post-dominates node 10. These results are useful already; an experienced developer might have already concluded that statement 5 is where the problem is.

In the re-slicing step, the hybrid slice is computed using the pruned static slice as well as the control-flow graph. In Figure 2, because nodes 11 and 12 are excluded, the assignment at node 12 can never affect the slicing criterion. In addition, as one branch of node 2 has been pruned, the assignment at node 1 can no longer affect the slicing criterion (the definition is "killed" by the assignment at node 3). As a result, the hybrid slice is {2, 3, 4, 5, 6, 7, 8, 9, 13}. Compared with the original static slice, the hybrid slice contains three statements fewer (namely, statements 1, 11, and 12). This reduction is achieved with purely the position of the executed logging statement. Moreover, the observed value in the logs can be picked up by a constraint solver, which potentially refines the slice further, as is done in SherLog [7]. In the example, using the value exposed at node 10 (i.e., `a`, which is 0), node 9 can be excluded too. Investigating the utility of a constraint solver is one of our future works.

Figure 3 illustrates the relationship between logs and the different kinds of slices (namely static, dynamic, and hybrid slices) with respect to a given slicing criterion. Static and dynamic slices are represented by squares with dashed borders, since AutoLog does not directly present the entire static slice, nor does it attempt to capture the dynamic slice. Obviously, hybrid slice is a subset of static slice and a superset of dynamic slice. The common property of these slices is that they all contain statements that may affect the slicing criterion. This property does not hold for logs, since there may be a lot of logs that are irrelevant to the failure. The shaded rectangle represents the intersection of dynamic slice and logs[1]. Only the logs in this rectangle will be presented to the developer, aligned with statements in the static slice. In other words, AutoLog intrinsically solves the problem of information overloading caused by irrelevant logs.

However, the sparsity of logs in practice might lead to information "underloading" instead, that is, there are missing logs that could have helped to pinpoint the bug. Conceptually, this happens when the root cause is not "covered" by the existing logs, as shown in Figure 3. A developer might add new logging statements, and then rerun the test. In log refinement, AutoLog automates this step by proposing and adding new logging statements. The net effect is to enlarge the set of logs (i.e., the lower right rectangle in Figure 3), while shrinking the hybrid slice towards the dynamic slice to cover the failure's root cause (RC). Since the dynamic slice consists of all the statements that are actually relevant to the failure, it must contain RC. In addition, as the hybrid slice (HS) is a superset of the dynamic slice, it certainly contains RC, too. When RC is not contained in the highlighted logs (HL), it must belong to the set $R = HS - HL$.

AutoLog iteratively selects some locations from R to insert new logging statements and re-executes the program to generate new logs. We follow a group of heuristic rules, with the goal to approach the root cause quickly with fewer logs:

Rule of loop avoidance. If a new logging statement is inserted in a loop, then it is likely to be executed for multiple times, which generates bloated log files. This rule avoids inserting logging statements in loops. More specifically, AutoLog calculates the depth of nested loops where candidate statements reside. Then the statements are sorted by the depth in an ascending order.

Rule of control-flow uncertainty reduction. During debugging it is often helpful to know whether certain statements have been executed in the failing run. In order to reduce the uncertainty of control-flow, AutoLog inserts new logging statements at the statements (in R) that **may** have been executed. Moreover, AutoLog prefers to choose the statements with higher power of uncertainty reduction. As previously described, the execution of one statement can be used to determine a group of statements which must have been executed (denoted as M) as well as a group of statements which must not have been executed (denoted as MN). When applying this rule, AutoLog first calculates the number $|M \bigcup MN|$ for each candidate statement to represent its power of uncertainty reduction and then sorts the candidates by the number in a descending order. In our example, the number $|M \bigcup MN|$ for nodes 5, 6, 7, 8, 9 is 2, 2, 4, 4, 4, respectively[2]. Therefore, AutoLog ranks nodes 7, 8, and 9 higher than nodes 5 and 6.

[1]More exactly the shaded rectangle represents the logging statements that are control-equivalent to some statements in the dynamic slice.

[2]For instance, for node 7, M is {5, 6} and MN is {8, 9}, thus its $|M \bigcup MN|$ is 4.

```
1: function RefineLogInMethod($method$)
2:     for all $stmt_i \in method.stmts$ do
3:         calculate its depth of loops, $d_i$
4:         calculate its number $n_i = |M_i \bigcup MN_i|$
5:         calculate its number of interesting variables, $m_i$
6:     end for
7:
8:     $NumStmts \leftarrow method.stmts.size$
9:     sort $method.stmts$ by $d_k$, $1 \leq k \leq NumStmts$, in an ascending order
10:    sort the tying statements in $method.stmts$ by $n_k$, $1 \leq k \leq NumStmts$, in a descending order
11:    sort the tying statements in $method.stmts$ by $m_k$, $1 \leq k \leq NumStmts$, in a descending order
12:    choose the top one statement, $topStmt$
13:    compute the interesting variables at $topStmt$, $vars$
14:
15:    create a new logging statement $logStmt$
16:    $logStmt.location \leftarrow topStmt$
17:    $logStmt.variables \leftarrow vars$
18:    insert $logStmt$ into $method.code$
19: end function
```

Figure 4: Algorithm of log refinement for one method

Rule of value uncertainty reduction. If a logging statement resides at a location at which many relevant variables are accessible, then it can reveal valuable information of runtime program state by printing the values of these variables. When applying this rule, AutoLog first calculates the set of visible interesting variables for each candidate statement. A variable is said to be interesting if it is in the data flow set before the statement in the data flow analysis during program slicing, that is, its value may be relevant to the slicing criterion. Then AutoLog sorts the candidates by the number of visible interesting variables in a descending order. In the example in Figure 2, since the interesting variable `tmp` is accessible at nodes 7 and 8 (but not at node 9), AutoLog prefers to choose either of nodes 7 and 8 as the candidate location to insert a new logging statement.

AutoLog applies the rules in the order they are described[3] and selects the location to insert the new logging statement from the top statement(s) for each method. If there are multiple top statements, AutoLog randomly chooses one of them. Therefore, in each iteration AutoLog inserts at most **one** new logging statement in each method to control the total amount of logs that will be generated. We also take care to print variables that have already been initialized at the new logging points, based on data flow information generated during program slicing. Figure 4 shows the algorithm of log refinement for a method.

Trade-offs. In theory, slices are computed using both control and data dependencies in order to conservatively include all the statements which may cause the failure. In practice, as discussed in [4], many bugs can be discovered by exploring data dependency. Moreover, control dependency can be converted to data dependency of the corresponding branching statements in structured programs. Therefore, AutoLog makes the trade-off between precision and efficiency, in that it does not use control dependency during slicing. As a result, the slices may fail to include clues to bugs, but the size of slices can be reduced. In case the developer finds that a control dependency is too important to ignore, she can manually specify the corresponding branching statement and variable as slicing criterion, and re-launch AutoLog.

The hybrid slicing algorithm of AutoLog is inter-procedural, that is, it will track data dependencies across procedure boundaries by following calling relations. When performing slicing in a method `m`, AutoLog will explore forward into the relevant methods called by `m` and backward into the methods that call `m`. The inter-procedural slicing is necessary for AutoLog to capture the root cause which does not reside in the same method with the error-reporting logging statement. In order to allow the hybrid slicing to handle large-scale systems, we impose two restrictions on the slicing procedure:

- In the forward exploration, the slicing procedure does not explore further along a call chain if a cycle is encountered (i.e., there are recursive calls).
- We augment each logging statement with an extra statement to record the stack trace. In this way, we can obtain the exact runtime stack trace of the starting log. Therefore, when performing backward exploration, the slicing procedure only has to explore the methods along the recorded stack trace, instead of all the call chains leading to the current method. Note that we just have to record the stack trace during the first execution in which the failure is reproduced. Once the failure has been observed, we can turn off the stack trace recording to reduce overhead in subsequent re-executions.

4. PRELIMINARY CASE STUDY

We have implemented a prototype of AutoLog on top of the Soot framework[4] which supports various program analyses on Java programs. We have not implemented the offline part that prepares the slice-DB. Rather, when performing log slicing, the slice is computed from scratch.

We now describe a proof-of-concept case study on Apache Hadoop Common, the core sub-project of the Apache Hadoop distributed computing framework[5].

In the case study, we have used the prototype to debug a reported bug[6] for Hadoop Common version 0.19.0. The bug manifests itself as a reproducible test case failure caused by an `IOException`. The bug is in the method `listStatus` of class `SequenceFileInputFormat`, a subclass of class `FileInputFormat`. The correct behavior of the method is to return the list of files contained in the variable `file`, if `file` is a directory. However, if one of the files is indeed a directory, rather than returning the status of that directory, the buggy implementation returns a new file object with attribute `isdir` as true.

The exception is thrown from method `getSplits` in class `FileInputFormat`. Before splitting files, the method checks whether the files, retrieved via a call to method `listStatus`, are actually files rather than directories. The tricky part is that class `FileInputFormat` has its own implementation of method `listStatus`, which has been overridden in class `SequenceFileInputFormat`. Thus the method call to method `listStatus` in method `getSplits` have two possible targets, and only when the underlying object is an instance of class `SequenceFileInputFormat`[7], the `IOException` will occur.

[3]Specifically, the application of a rule may only change the order of the candidate statements that have been ranked the same by the previous rule.

[4]http://www.sable.mcgill.ca/soot/

[5]http://hadoop.apache.org/

[6]https://issues.apache.org/jira/browse/HADOOP-3946

[7]Besides `SequenceFileInputFormat`, there are several subclasses of `FileInputFormat`.

In the original log file generated by the failed test run, there were 86 lines of logs. AutoLog highlighted 17 logs, containing the most relevant logs printed in method `listStatus` of class `FileInputFormat`. The rest of the logs were mostly about the progress of map/reduce tasks, which were irrelevant to the exception. However, the 17 highlighted logs might be insufficient for finding the bug, and even a bit misleading. Because the last line of log was printed in method `listStatus` of class `FileInputFormat`, developers might conclude that the erroneous files were returned from this method. Since the implementation of this method is complex, developers might waste a lot of time and fail to find the real bug in method `listStatus` of class `SequenceFileInputFormat`. Therefore, we used the prototype to insert new logging statements and re-ran the test case. After two iterations of refinement, the logs successfully covered the root cause. The last two lines of logs were printed from method `listStatus` of class `SequenceFileInputFormat` and method `isDir` of class `FileStatus`, which made the bug quite obvious.

Although the prototype succeeded in revealing the bug, the final log file contained as many as 1778 lines of logs. It was mainly due to our unsophisticated rules of log refinement. For example, the current rule only tries to avoid inserting logging statements in explicit loop structures in the scope of a method. However, in the case study, a number of logging statements were inserted into methods that were called from within loops in their callers.

The case study was conducted on a Linux server, which has a 2.33GHz quad-core CPU and 16 GB main memory. The average runtime of each iteration (including log slicing and log refinement) is about 11 minutes. For Hadoop Common, which has more than 143K lines of code, the runtime is probably acceptable. Our existing prototype does not allow easy estimation of how much time can be moved to offline. However, we are optimistic that a good portion will be.

5. DISCUSSION

The exercise of debugging asks the programmer to act as a detective. Having arrived at the crime scene, she needs to reconstruct the sequences of events, reasoning about the "why" and "how" along the way. Building a time machine that allows replaying the entire execution path represents one type of tools. The spectrum includes tools such as R2 [3] that relies on instrumentation at the boundary of non-determinism, to SherLog [7] that infers the paths with the combined knowledge of runtime logs and program structure (via constraint solving).

Replaying tools narrow down the search space to concrete execution instances, but do not solve the problem of constructing the path that connects failure site to root cause. In addition, knowing the entire execution history may not be necessary.

At the other extreme of the spectrum, there are bug detection tools that try to directly identify common mistakes (e.g., use-after-free [1]). These faulty patterns are shortcuts to directly flag not only the manifested bugs but also the potential ones. However, we argue that discovering causality interactively is the most common case, and needs better tools. A natural starting point is to leverage the technique of slicing [5].

AutoLog is somewhere in between. We target the scenario of interactive in-house development, in which quick turnaround time is critical. This is achieved by splitting offline static slice preparation from online log slicing and refinement. We share the same philosophy as LogEnhancer [8], in that the existing logs are treated as heuristics and starting points, and a tool is free to modify them, as well as to insert new ones.

Our future work has a few focuses:

1. For log slicing, simply pruning the static slices with log positions, performing hybrid slicing, and calling a constraint solver incrementally improves coverage, with increasingly higher cost and longer delay. The trade-offs there are complex, and we plan to quantify them with a set of real examples.

2. The heuristics used in the current log refinement algorithm is neither sophisticated nor well tested. Again, this requires in-depth study over real examples.

3. Once the programmer formed concrete clues and installed new fixes, the static slices need to reflect these changes. Improving the productivity requires us to recompute the slice-DB incrementally and efficiently.

4. The above process works well when non-determinism is absent. This isn't the case for many concurrent programs. We see the role of replay tools to remove the randomness so that AutoLog can do its work with as little as (and ideally no) uncertainty. How to leverage these techniques is still an open question.

5. An orthogonal but important question is to understand and improve the quality of the logs. If and when AutoLog is continuously applied throughout the development process, it is possible to handle and manage the logs better. As we mentioned earlier, the framework of AutoLog can already detect redundant logs.

6. REFERENCES

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

[2] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. SAS'03, pages 1–18, Berlin, Heidelberg, 2003. Springer-Verlag.

[3] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. In *OSDI'08*, pages 193–208, Berkeley, CA, USA, 2008. USENIX Association.

[4] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM.

[5] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, 1994.

[6] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[7] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. ASPLOS '10, pages 143–154, New York, NY, USA, 2010. ACM.

[8] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. ASPLOS '11, pages 3–14, New York, NY, USA, 2011. ACM.