Debugging with Online Slicing and Dryrun

Cheng Zhang[1], Longwen Lu[1], Hucheng Zhou[2], Jianjun Zhao[1], Zheng Zhang[2]
[1]Shanghai Jiao Tong University, [2]Microsoft Research Asia

ABSTRACT

Efficient tools are indispensable in the battle against software bugs during both development and maintenance. In this short paper, we introduce two techniques that target different phases of an interactive and iterative debugging session. To help fault diagnosis, we split the costly computation of backward slicing into online and offline, and employ incremental updates after program edits. The result is a vast reduction of slicing cost. For the benchmarks we tested, slices can be computed in the range of seconds, which is 0.3%~5% of the unmodified slicing algorithm.

The possibility of running slicing in situ and with instant response time gives rise to the possibility of editing-time validation, which we call dryrun. The idea is that a pair of slices, one forward from root cause and one backward from the bug site, defines the scope to validate a fix. This localization makes it possible to invoke symbolic execution and constraint solving that are otherwise too expensive to use in an interactive debugging environment.

1 INTRODUCTION

Debugging is both black art and tough business. There does not appear to be a silver bullet that addresses all the major pain points of a developer. In part, this is because debugging itself involves phases that are qualitatively different.

Consider a typical debugging session after a bug report is submitted. Assuming that the fault site has been clearly identified, the first phase is to trace back towards the root cause by reasoning about the causality chains. This phase may itself be iterative, as the developer adds new logs and/or mutates inputs to collect additional evidence. Once the root cause is discovered, the developer must select among a number of candidate fixes and apply the best one. Typically, the patched program needs to be compiled and re-run against test cases. Before the patch is officially released, peer review is required.

The most expensive resource in this whole process is the human time. However, for large software packages and especially in the cloud computing era, securing hardware resources and perform re-deployment, re-execution and log collection also entail non trivial cost.

It is evident that different tools are needed in these different phases, although they share the common objective to reduce primarily the human time. For example, during the phase of analyzing the root cause, the most effective tool should address the tedious issue of causality reasoning. Today, most if not all debugging start with printed logs. Logs can be valuable as they represent a snapshot of execution path as well as selected concrete values of certain states. Logs, however, can be both incomplete and voluminous. The fact that reasoning with logs remains the dominant practice suggests that effective and practical tools in causality reasoning have not matured to the point of adoption.

The best tool to identify causality chains is slicing, which constructs a collection of codes called a *slice* [9]. Basically, starting from a selected statement, a *forward* slice is the collection of program statements that this statement has effect on, and a *backward* slice is the collection of program statements that affects this statement. Taken together, they are powerful concepts to reveal data dependency. Slicing can be further divided into *dynamic* and *static* slicing. Dynamic slicing [1, 17], which records the exact execution path, requires heavy instrumentation, imposes high runtime overhead, and is therefore seldom used in practice. Static slicing is typically done at compile time and has zero runtime overhead. Lacking enough dynamic information, however, static slicing can be ambiguous.

In our previous work [16], we show that logging and slicing can complement each other: log entries can prune uncertainties of slices; slices, on the other hand, represent the developer's focus in her reasoning about the bugs, and thus can prune irrelevant logs. Moreover, new log entries can be suggested or

even automatically inserted to improve slicing precision, and help to zoom in the root cause in an interactive manner.

While the approach appears promising, it has made the assumption that static slicing can be integrated in the interactive and iterative debugging process, instead of being a one-time offline process; any new updates will render previous computed slices stale. Such updates should be done in situ and instantaneously. Unfortunately, our previous experiments have shown that employing classic slicing method is prohibitively slow: one of the Hadoop bug that we identified took 10 minutes to produce the slice.

Our first contribution is the implementation of *online* slicing, bringing down slicing updates to seconds, instead of minutes. The magnitude of performance improvement makes it possible to use slicing as a primary tool at debugging time, in a responsive and interactive manner. This is achieved by splitting slicing into offline and online portion, and employs a combination of techniques to make updates incremental and local.

Once a fix is applied, the updated software must be thoroughly tested. Doing so requires both resource and also human time. Worse yet, if a partial fix is prematurely released, the cost is even higher; for security bug, the incomplete fix actually reveals the vulnerability even further. One would argue that the process of understanding the root cause and devising the fixes has intrinsically performed the validation. However, as bugs become more complex, so do their patches. Nearly 70% of patches are buggy in their first release [8]; and 14.8% ~ 24.4% of fixes released out in large OSes are reportedly bad patches [11]. Clearly, we need some way to validate the patches as early as possible.

In the same spirit of bringing slicing online, we propose the idea of editing-time patch validation using a concept called *dryrun*. Conceptually, the forward slice rooted at the (supposedly) root cause will intersect the backward slice rooted at the bug site. The resulting set, called *RB-scope*, defines the impact scope of the patch. The basic idea is that, for a patch to be effective it must satisfy two conditions. First, it must be located inside the RB-scope. Second, it should disable conditions that has led to the original failure. Thus, by using symbolic execution and constraint solving (and possibly symbolic model checking), we can perform a step of patch validation even before the code is compiled. This is a concept that is still being developed. However, manual inspection upon a few bugs reveals that this may be a new and promising direction.

The rest of the paper is organized as follows. Section 2 gives an overview, followed by description of online slicing and dryrun validation. We present preliminary result in Section 3, along with a short discussion. Section 4 covers related work, and we conclude with Section 5.

2 System Architecture

Our system provides a set of fundamental services to support bug diagnosis and patch validation (Figure 1). These services, in turn, rely on a number of building blocks. By themselves, these building blocks perform classic static analysis. However, we pay particular attention in making them rapidly responsive in an iterative *and* interactive debugging session.

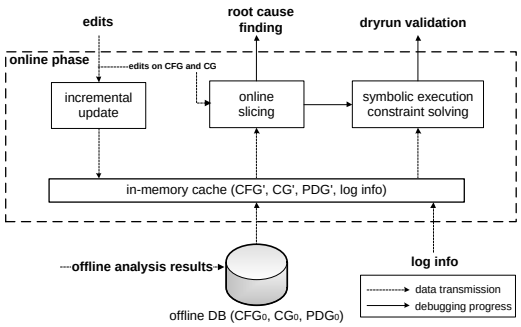

Figure 1: System overview.

Our system is split into offline and online portion. The offline engine performs expensive but (more) precise static analysis, and stores the results in a database, which are loaded into an in-memory cache at the beginning of the session. Online slicing (see Section 2.1) is triggered by developer for bug diagnosis or new edits for patches, the results can be committed to the database at user's control. Online slicing also feeds the slices to the dryrun engine, which calls symbolic execution and/or constraint solver to perform in-edit validation (see Sec-

tion 2.2). Finally, log information are collected during runtime to help refine the precision.

2.1 Online Slicing

In our system, slicing plays an important role, it needs to be as fast as possible, and at the same time maintain precision. Program slicing essentially identifies the slice of a variable at given program points by analyzing both data and control dependencies from program dependence program (PDG) [4]. Data dependency can be computed from reaching definitions analysis, while control dependency can be derived from control flow graph (CFG) by building post-dominator trees. In our current approach, we focus on the data dependencies and leave control dependencies for future work, but compensate the loss of precision partially by incorporating log entries. Compared with computing the CFGs and call graph (CG), data flow analysis is generally much more expensive and thus takes a dominant part of the runtime of slicing.

As is the case for most static analysis, speed and precision is a difficult tradeoff: improving precision requires more extensive analysis, and thus takes longer to complete. Therefore, in order to achieve online slicing, we separate the analysis into online and offline parts. Intuitively, the most time-consuming but relatively stable analysis is computed once offline, and the results are stored in the database and loaded on-demand.

In our system, we put the entire program intermediate representation as well as the corresponding CFGs and CG into disk in binary format like ELF[1]; alternatively we can store the semantic information in BDB thus using bddbddb [10] to query the result. More importantly we also put the analysis results of alias, reaching definitions, as well as the resulting PDG into the offline database. We do not keep slicing results since the slicing criterion is selected by the user and unknown a priori.

Given up-to-date PDGs and CG, computing both backward slice and forward slice is straightforward and cheap. This happens at the beginning of the debugging session. The challenge arises when new edits are introduced.

[1] `http://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

We investigated and implemented two approaches, demand-driven [3] and incremental [5] data flow analysis. Due to space limitation, we will describe the second approach. We first decompose edits into the set of *structural* ones that changes CFGs and CG of the program, and *non-structural* ones that do not. Updating CFGs and CG is generally cheap. We then incrementally re-compute the data flow analysis. The data flow algorithm is iterative in nature, potentially touching the entire program variables and points. Provided that the edits are local, however, by propagating those changes outwards from edit sites, the data flow analysis can typically arrive fix point earlier [5].

As described in our previous work [16], slicing and log analysis can complement each other. In particular, online slicing can leverage log entries to refine related parts in graphs (both CFGs and CG) (e.g. which branch has been taken).

2.2 Dryrun Validation

Once a patch is applied, it needs to be validated. Existing practises entail a chain of events: compilation, deployment and running against test cases, submission for peer reviews and then release. Some of the steps are costly and time-consuming. Worse yet, incomplete patches will trigger the whole debugging processes all over again, while exposing vulnerability in between.

Dryrun follows our principle of applying static analysis in situ, but doing so instantaneously at the bug fixing time as soon as an edit is inserted. Its goal is to do as much validation as possible before the code even starts to get built.

In a nutshell, the intuition is that the intersection of forward slice at the root cause with backward slice starting at the bug site has defined the scope of bug fixing and validation. We call the intersection of the two slices the *RB-scope*. This "localization" limits the scope of applying expensive but powerful static analysis such as symbolic execution and constraint solving, and does so only on relevant states identified by the slices.

The basic intuition behind dryrun is that, for a patch to be valid it must satisfy two conditions. First, it must lie in the RB-scope described above. Second, the patch must disable some old conditions and/or enforce the right constraints, such

```
1  /* fs/xfs/linux-2.6/xfs_acl.c */
2  static struct posix_acl *
3  xfs_acl_from_disk(struct xfs_acl *aclp)
4  {
5    const size_t size;
6    /* New patch */
7  - int count, i;
8  + unsigned int count, i;
9
10   count = be32_to_cpu(aclp->acl_cnt);
11   /* Partial Patch */
12 + if (count > XFS_ACL_MAX_ENTRIES)
13 +   return ERR_PTR(-EFSCORRUPTED);
14   acl = posix_acl_alloc(count, flag);
15   ...
16   for (i = 0; i < count; i++) {
17     /* initialize the content of acl here. */
18     acl_e = &acl->a_entries[i];
19     ...
20   }
21   ...
22 }
23
24 /* fs/posix_acl.c */
25 struct posix_acl *
26 posix_acl_alloc(int count, gfp_t flags)
27 {
28   size_t size = sizeof(struct posix_acl) +
29     count * sizeof(struct posix_acl_entry);
30   struct posix_acl *acl = kmalloc(size, flags);
31   ...
32 }
```

Figure 2: Dryrun example from XFS filesystem.

that the bug will not manifest. A comprehensive patch should disable all the violation of constraints, whereas a partial patch disables only part of them.

We will illustrate the idea of dryrun with a bug of integer overflow in the Linux kernel's XFS filesystem[2], shown in Figure 2. Note that *count* is converted from $aclp \rightarrow acl\_cnt$, which is directly read from disk in line 10 thus can be any signed value. Given a large *count* with positive value (e.g. $0x080000FF$) and both *sizeof* operations returns 16, integer overflow would occur at the multiply operation (line 27 in *posix_acl_alloc*). As a result, *kmalloc*() in line 28 allocates a smaller memory (i.e. 141) than *count* dictates. It is worth to note that the two functions are located at different files under different directories.

A patch was devised (line 12-13) that enforces a upbound of *count*. The problem is that the attacker can submit a negative value, completely bypass the sanity check and yet still causes integer overflow, leading to possible memory corruption.

[2] https://bugzilla.redhat.com/show_bug.cgi?id=773280

We note that with program analysis, at the time when *count* is used, we can derive a constraint: $count < SIZE\_MAX/sizeof(struct\ posix\_acl\_entry)$ (from *kmalloc*()), and $count > 0$ (from the loop constraint in line 16). A forward symbolic execution along the slice of *count* from line 12 shows that, without the patch, no constraint is imposed. Further, it indicates that the patch still leaves one side of the constraints unsatisfied, indicating that the patch is incomplete. An additional patch is added at line 7 ~ line 8. As evident in this example, the RB-scope is small, consisting the intersection of forward slice on *count* at line 10, and the backward slice on *count* at line 16.

We have also found that in some other cases symbolic execution and constraint solving are not enough, since multiple paths (and hence slices) can reach the bug site in non-deterministic fashion. As such, symbolic model checking will show a concrete order of interleaving. The sweetspot is when the distance between root cause and bug site is sufficiently far such that pure mental reasoning is hard, and yet close enough such that using slicing, thus local application of symbolic execution, constraint solving and model checking is cheap and effective.

3 Implementation and Preliminary Results

We have implemented most of the components of online slicing as described in Section 2.1. For the time being, we used an in-memory data structure to implement the database. We leveraged Soot analysis framework[3] for constructing CFG, CG, and PDG.

We have integrated the slicing with log pruning [16]. As program edits may invalidate the log information, log pruning and log-based hybrid slicing are only performed in the first iteration of slicing. How to reuse valid logs across edits is one of our future work. We are in the process of investigating more details of dryrun, and integrating symbolic execution, constraint solving and symbolic model checking.

We use Apache Velocity[4] (88K LOC) and Apache Hadoop Common[5] (174K LOC) as the

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

[4] http://velocity.apache.org/

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

benchmarks. We selected four bug reports for evaluation (three from Velocity and one from Hadoop Common). The selection criterion is based on the reproducibility of failures, clear identification of root causes, and the size of patches.

The evaluation on Velocity is performed on a typical developer's machine with 2.80GHz dual-core CPU and 4GB memory, running Windows 7. The Hadoop case is run on a machine with 96GB memory, running Ubuntu Linux, since its analysis results are far bigger.

Program	Bug No.	Offline	Initial Slicing	Update & Slicing
Velocity	625	78.63	0.12	0.22
	685	71.28	3.80	3.71
	651	74.42	1.87	2.06
Hadoop	4288	280.14	14.91	15.06

Table 1: Average run time of each step (sec.)

3.1 Results

We performed the following three steps and report their running times. 1) Offline analysis; including building CFGs and CG, exhaustive data flow analysis, and PDG construction. This the most time-consuming step. 2) Online backwards slicing; to prepare the developer for causality reasoning. The slicing is rooted at the bug site, and takes the PDGs and CG that is computed offline. 3) Slicing after edits; it simulates the impromptu program edits, based on the patches in the bug report. A patch for bug fix may not represent the actual sequence of edits during the debugging session, but we believe that this is a reasonable approximation. We decompose the changes into elementary edits and then perform the corresponding incremental updates on the information computed by the offline step. The updated PDGs and CG drive the re-computation of online slicing: backward slicing to continue aiding causality reasoning, and forward slicing in helping dryrun-based validation.

Table 1 shows the analysis time of each step. As expected, the offline part is the most time-consuming, whereas the online part, especially the incremental update, is significantly faster. This is mainly due to the fact that the patches in these bug cases are local and thus the incremental update can complete very quickly. Slicing has to transitively track dependencies (some of which are inter-procedural), they are more expensive than the incremental update, but they are quick enough to support interactive debugging, where developers are likely to make code inspection or edits between iterations of slicing.

Since Hadoop Common is much larger and complex than Velocity, its runtime is also much longer. The call graph in Hadoop is 800% more than Velocity (408,038 versus 50,271), thus slicing has to take into account more inter-procedure dependencies.

3.2 Discussion

Our results are encouraging. However, there are many places to improve, some of them more immediate than others. The use of a high performing disk-based database will eliminate the need of large memory, especially for large-scale packages such as Hadoop. Similarly, slices could have been precomputed and/or cached. Also, to fully develop the idea of dryrun, we need to look into more bug cases.

We have also learned a number of hard lessons, resolving them comprise our longer term agenda. Most of the lessons belong to the classic issue of trading off precision with cost. The problem is all the more significant for us, since running analysis is directly on the critical path. Controlling the aggressiveness of slicing, for instance the choice of when to use context-sensitive slicing, whether control dependencies are important (several bug cases show that they do), may take the path of learning the surface features from empirical data in order to be adaptive.

4 Related Work

There is a large body of research work related to our system, in the field of failure diagnosis and test-driven development. In terms of failure diagnosis, there are broadly speaking three major directions: execution comparison [7, 8, 15], source code and runtime information correlation [12, 13, 14, 2], and slicing [9, 1]. All these works entail a tradeoff between cost and precision in various ways. Our general approach combines runtime information to make slicing more precise. However, we recognize the need to integrate log-assisted slicing into an interactive and iterative debugging experience, and hence the challenge of handling slicing quickly.

Our technique of performing online slicing borrows ideas from incremental data flow analysis [5] and demand driven data flow analysis [3]. Most of these ideas were proposed nearly two decades ago, and were never tested in real settings. The complex landscape of modern software as well the advancement of hardware has made this line of work both relevant and practical. As far as we know, this is the first serious attempt to implement the algorithms. As we have shown, while our results are encouraging, by exercising the algorithms in real, we have uncovered a number of new research problems, such as how to combine the algorithms in the best synergistical way, and how to improve precision while keeping the cost down for practical use.

Development and testing are usually thought as two closely related but temporally separate phases. However, there have been proposals that integrate the two more closely, such as continuous testing [6]. The idea is to integrate testing harness in IDE so as to run test asynchronously, and notify developers if regression exists. The hope is to improve the quality of a programming task. Dryrun shares the same philosophy that validation should happen as early as possible. However, dryrun focuses directly on validating whether a fix is sound, and does so by using an array of powerful and sophisticated static analysis tools such as symbolic execution and constraint solver, and yet limit their expense by defining the scope where the tools are applied. This novel approach potentially opens a new direction where advanced static tools can be applied.

5 Conclusion and Future Work

Given that developers spend much of their time finding bugs and fixing them iteratively and interactively in an IDE, we believe it's time to bring more sophisticated analysis tool into editing time. We developed online slicing to rapidly response to program change, and further leverage its power to perform editing-time validation of fixes. While the preliminary results are encouraging, much work remains. Our future work includes optimizations to online slicing to make better tradeoffs between precision and cost strategically, and to develop dryrun methodology fully.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, 1990.

[2] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *EuroSys*, 2011.

[3] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *TOPLAS*.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 1987.

[5] L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Trans. Softw. Eng.*

[6] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA*, 2004.

[7] W. N. Sumner, T. Bao, and X. Zhang. Selecting peers for execution comparison. In *ISSTA*, 2011.

[8] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *ASPLOS*, 2009.

[9] M. Weiser. Program slicing. In *ICSE*, 1981.

[10] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04*, 2004.

[11] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram. How do fixes become bugs? In *SIGSOFT FSE*, 2011.

[12] 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.

[13] D. Yuan, S. Park, and Y. Zhou. Characterising logging practices in open-source software. In *SIGSOFT ICSE*, 2012.

[14] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *ASPLOS*, 2011.

[15] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys*, 2010.

[16] C. Zhang, Z. Guo, M. Wu, L. Lu, Y. Fan, J. Zhao, and Z. Zhang. Autolog: facing log redundancy and insufficiency. In *APSys*, 2011.

[17] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *SIGSOFT FSE*, 2006.