MoonBox: 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. In this short paper, we introduce two techniques that target different phases of an interactive and iterative debugging session. To make slice-assisted log analysis practical to help fault diagnosis, slicing itself must be done instantaneously. We split the costly slicing computation 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 0.02%˜6.5% 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 performing re-deployment, re-execution and log collection also entail non-trivial cost.

It is evident that different tools are needed in these different phases. For example, when 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, which can be both incomplete and voluminous.

The best tool to identify causality chains is slicing, which constructs a collection of codes called a *slice* [13]. 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 causality. Slicing can be divided into *dynamic* and *static* slicing. Dynamic slicing [1, 21], which records exact execution traces, usually 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 [20], 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. This process can be iteratively applied to identify the root cause.

While the approach appears promising, in order to be practical in the interactive debugging context, slices must be computed and updated in situ and instantaneously when new edits are admitted. 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 [12]; and 14.8% ~ 24.4% of fixes released out in large OSes are reportedly bad patches [15]. 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 *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. This localization makes it possible to employ the otherwise expensive symbolic analysis to validate the fix even before the code is compiled. For example, from the failure site we can perform two backward symbolic executions to derive bug-triggering predicates at the root cause, one in the original and another in the patched code. If a constraint solver finds the intersection of the two predicates solvable, then the fix is partial. 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. This localization reduces the cost of 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.

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.

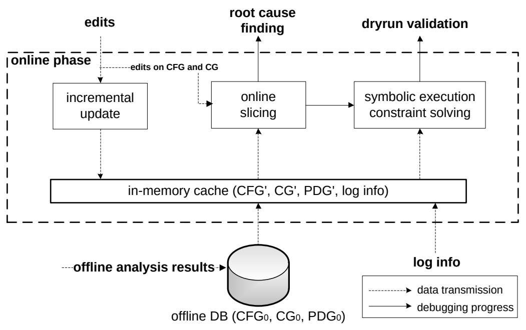


Figure 1: System overview.

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 Section 2.2). Finally, log information are collected during runtime to help improve 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 graph (PDG) [6].

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; alternatively we can store the semantic information in BDD thus using bddbddb [14] 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.

We investigated and implemented two approaches, demand-driven [5] and incremental [10] 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 [10].

As described in our previous work [20], 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

Dryrun is an application of online slicing, but targets the problem of patch validation. The goal is to compute a counter example for incomplete patch at editing time, establishing a defense line as early as possible.

The foundation of dryrun is RB-scope, which is the the intersection of forward slice at the root cause with backward slice starting at the bug site. In general, the patching point is considered as the root cause and thus it must be enclosed by the RB-scope.

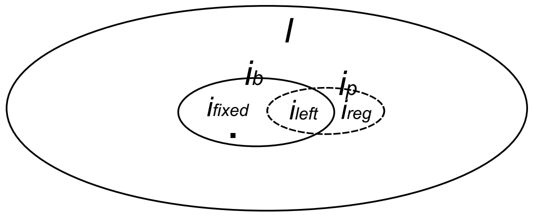
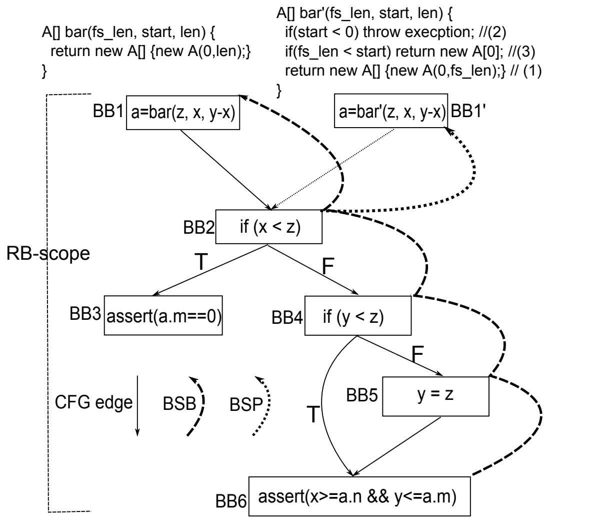


Figure 2: Bug-triggering input set and their interrelations.

Figure 2 illustrates the main concepts. I denotes the total input set starting at the root cause. The concrete value that triggers the failure (represented by the solid dot) must belong to the bug triggering input set i_b, which reaches the failure site. A correct patch removes i_b, without introducing new bugs. An incomplete patch (with bug triggering input set i_p) may remove only a subset of i_b, denoted as i_{fixed}, leaving an non-empty subset yet to be fixed (i_{left}), and/or introducing regression (i_{reg}).

i_b is represented by conditions known as the weakest precondition [4], and can be computed using backward symbolic execution [9] from bug site to root cause. When a patch is introduced, similar process yields i_p, from which we can compute i_{fixed}, i_{left}, and i_{reg}. The basic idea of dryrun is to monitor the above sets *online* as patches are derived. These sets are symbolic, and can be fed into a constraint solver. In particular, a concrete solution to i_{left} identifies a counter-example of an incomplete patch.



A[] bar(fs_len, start, len) {
return new A[] {new A(0,len);}
}
A[] bar'(fs_len, start, len) {
if(start < 0) throw execption; //(2)
if(fs_len < start) return new A[0]; //(3)
return new A[] {new A(0,fs_len);} // (1)
}
BB1
a=bar(z, x, y-x)
a=bar'(z, x, y-x)
BB1'
BB2
if (x < z)
RB-scope
T
F
BB3
assert(a.m==0)
BB4
if (y < z)
F
CFG edge
BSB
BSP
T
BB5
y = z
BB6
assert(x>=a.n && y<=a.m)

Figure 3: Dryrun Illustration. RB-scope for assertion at BB6, and BSB and BSP stand for paths for backward symbolic execution for original buggy program and the patch program, respectively. (1), (2) and (3) are the corresponding fixes patched one after another. Expressions *a.m* and *a.n* represent accesses to fields of variable *a*.

We illustrate the idea of dryrun with a real bug[1] in Apache Hadoop Common. There are three different patches (even more considering updates of unit tests) in one week. For simplicity, we abstracted the control flow graph with slices shown in Figure 3. There are essentially three assertions: $y \leq a.m$, $x \geq a.n$, and $a.m == 0$. Taking $x \geq a.n$ for instance, its i_b, i_p, i_{fixed}, and i_{left} after the first patch are $\{x \geq z, x \leq 0, y \geq z\}$, $\{x \geq z, x \leq 0, y \geq z\}$, $\{\emptyset\}$, $\{x \geq z, x \leq 0, y \geq z\}$, respectively. Since i_{left} is not empty, dryrun identifies the patch as incomplete.

To make dryrun fast is challenging, as it not only depends on slicing, but also backward symbolic execution and constraint solving. We are reasonably optimistic because 1) it is known that distances between bugs and root causes are usually not large and 2) i_{left}, being a subset of i_b, is cheaper to solve than the latter.

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 sweet spot 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, leveraged Soot analysis framework[2] for CFG, CG, and PDG construction. We have integrated the slicing with log pruning [20]. As program edits may invalidate the log information, log pruning and log-based hybrid slicing are only performed in the first iteration of slicing. Section 3.2 discusses work in progress.

We select 10 bug reports from Apache Commons Math[3], Apache Commons Collections[4], Apache Velocity[5], and Apache Hadoop Common[6], as the benchmarks. The selection criterion is based on the reproducibility of failures, clear identification of root causes, and the size of patches. Our results of slices do not include code from libraries, as none of these bugs are library related.

3.1 Results

We performed the following three steps. 1) Offline analysis; including building CFGs and CG, exhaustive data flow analysis, and PDG construction. 2) Online slicing: backward slicing to continue aiding causality reasoning, and forward slicing in helping dryrun validation. The slicing takes PDGs and CG that are computed offline. This step also computes the RB-scope by intersecting the backward and forward slices. 3) Slicing after edits; it simulates the impromptu program edits, based on the patches in the bug reports, and the slicing on the updated program. Patches for bug fix may not represent the actual sequence of edits during debugging, but we believe this is a reasonable approximation.

Table 1 summarizes our results. As expected, the slices are significantly smaller than the program. This is due to several factors: 1) by definition, a slice

[1] `https://issues.apache.org/jira/browse/HADOOP-4677`

[2] `http://www.sable.mcgill.ca/soot/`

[3] `http://commons.apache.org/math/`

[4] `http://commons.apache.org/collections/`

[5] `http://velocity.apache.org/`

[6] `http://hadoop.apache.org/common/`

Program	Bug ID.	#BS	#FS	#RB	#Total	Offline	Initial Slicing	Update & Slicing
Math	744	4	2035	3	62952	102.73	0.07	0.05
	716	3358	3542	2235	63094	102.08	0.31	0.24
	567	477	515	258	45430	83.07	0.04	0.02
Collections	307	4244	4120	2548	26561	78.73	0.62	0.47
	299	4273	4152	2573	26539	77.20	0.75	0.60
	294	4245	4116	2546	26561	77.99	0.77	0.61
Velocity	625	2949	2997	1669	29260	129.99	1.56	1.35
	658	2931	2956	1652	29252	130.74	1.55	1.45
	728	2935	2959	1655	29260	134.89	1.47	1.28
Hadoop	4677	13339	19712	8858	173862	172.99	12.29	12.09

Table 1: Size of the programs, slice (BS for backwards and FS for forward), RB-scope, and run time of each step (sec.). The evaluation on Math and Collections is performed on a 2.80GHz dual-core CPU and 4GB memory, running Windows 7. The evaluation on Velocity is performed on a Linux server with 16G memory and the Hadoop case runs on a Windows server with 96GB memory.

only contains statements that may affect (or be affected by) the slicing criterion; 2) in our online slicing algorithm, we focus on data dependencies and ignore control dependencies, which may further reduces the slices. We note that the detailed treatments on special data dependencies, such as those caused by global variables and array accesses, can also affect the size (and precision) of the slices.

As the intersection of backward and forward slices, RB-scope is smaller than either of them (in most cases about 50% of the smaller slice). However, it is still significant. For instance, although the path of the Hadoop case includes a handful of basic blocks under manual inspection (see Section 2.2), RB-scope currently includes close to 9K statements. Perform backward symbolic execution is clearly too expensive. One of the reasons is that our inter-procedural slicing algorithm (based on [13]) is context-insensitive and conservative.

As expected, the offline part is the most time-consuming, whereas the online part, especially the incremental update, is significantly faster. As there still exist much room to improve, we are confident that our techniques can be used in the interactive debugging context.

3.2 Discussion and work in progress

Our preliminary results are encouraging. However, there are many places to improve, some of them more immediate than others. For instance, we are implementing an incremental version of the context-sensitive slicing algorithm [8]. We are also investigating how to reuse valid logs across edits. More importantly, we are going over a collection of bugs and understanding how to reduce RB-scope with sound heuristics, while integrating backward symbolic execution, constraint solving and symbolic model checking into our framework. In addition, we are studying techniques (such as [2]) to generate weakest preconditions for large RB-scopes.

We have also learned several hard lessons, resolving them comprises 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 introduce context-sensitivity, whether control dependencies are important (several cases show that they do), may take the path of learning surface features from empirical data in order to be adaptive.

4 Related Work

There is a large body of related work in the field of failure diagnosis and test-driven development. In terms of failure diagnosis, there are broadly speaking three major directions: execution comparison [11, 12, 19], source code and runtime information correlation [16, 17, 18, 3], and slicing [13, 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. More importantly, 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 online slicing borrows ideas from incremental [10] and demand driven [5] data flow analysis. Most of these ideas were proposed decades ago, and were rarely tested in real settings. The complex landscape of modern software as well the advance 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.

The bad fix problem is introduced and formalized by Gu et al. [7]. Their solution, FIXATION, is based on an adapted weakest precondition generation and forward symbolic execution. Dryrun shares the similar idea of representing bug-triggering inputs by weakest preconditions. However, FIXATION uses forward symbolic execution to compute bug fix coverage and uses test output to evaluate regression. In contrast, dryrun uses two rounds of backward symbolic execution to generate two versions of weakest preconditions which are used to detect the completeness of the bug fix and possible regression. Moreover, by focusing on the specific failure, dryrun can limit the symbolic execution in the RB-scope, potentially improving its efficiency.

5 Conclusion and Future Work

Given that developers spend much of their time finding and fixing bugs iteratively and interactively in IDEs, we believe it's time to bring more sophisticated analyses into editing time. We developed online slicing to rapidly respond to program changes, and further leverage its power to perform editing-time fix validation. 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.

Acknowledgment

This work was supported in part by National Natural Science Foundation of China (NSFC) grants 60970009, 60673120, and 91118004.

References

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

[2] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*, 2009.

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

[4] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[5] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *TOPLAS*, 1997.

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

[7] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *ICSE*, 2010.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *TOPLAS*, 1990.

[9] J. C. King. Symbolic execution and program testing. *CACM*, 1976.

[10] L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Trans. Softw. Eng.*, 1989.

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

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

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

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

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

[16] 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.

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

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

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

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

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