BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **20**, No 1
Sofia • 2020 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2020-0004

IDD – A Platform Enabling Differential Debugging

Martin Vassilev[1], *Vassil Vassilev*[2], *Alexander Penev*[1]

[1]*University of Plovdiv Paisii Hilendarski, 4002 Plovdiv, Bulgaria*
[2]*Princeton University, NJ 08544, USA*
E-mails: mvassilev@uni-plovdiv.bg vvasilev@cern.ch apenev@uni-plovdiv.bg

***Abstract*:** *Debugging is a very time consuming task which is not well supported by existing tools. The existing methods do not provide tools enabling optimal developers' productivity when debugging regressions in complex systems. In this paper we describe a possible solution aiding differential debugging. The differential debugging technique performs analysis of the regressed system and identifying the cause of the unexpected behavior by comparing to a previous version of the same system. The prototype, idd, inspects two versions of the executable – a baseline and a regressed version. The interactive debugging session runs side by side both executables and allows to examine and to compare various internal states. The architecture can work with multiple information sources comparing data from different tools. We also show how idd can detect performance regressions using information from third-party performance facilities. We illustrate how in practice we can quickly discover regressions in large systems such as the clang compiler.*

***Keywords*:** *IDD, differential debugging, functional regressions, performance regressions, complex systems, side by side debugging, interactive visual debugging.*

1. Introduction

Last decades software systems have grown in size and complexity. It is not uncommon to have a software system consisting of several million lines of code, which is developed by a community of hundreds of developers. Changes in one component might trigger undesired behavior elsewhere which is very challenging to isolate and reproduce [1]. Isolated minimal problem reproducers are often used as communication tokens between developers or development teams and shorten the ever-standing discussion whether it was a bug or a feature. Once this piece of information is available and the system regression evident the fix is often trivial.

Opposed to the past when hardware was more expensive than the manpower developing the software, the last decade has outlined that cheap hardware and expensive human development time are here to stay. This trend has set developer's productivity as a major field of interest for industry and academia. It has been widely accepted that most of the development time is spent in discovering and fixing

software problems in existing codebases causing expensive maintenance [2]. Debugging is a time consuming task, which is not well supported by existing tools.

There are two major types of problem discovery methods – static and dynamic analysis. Static analysis is limited as the program is inspected for common mistake patterns without being executed. Dynamic analysis can be divided into interactive and non-interactive. The non-interactive usually involves binary instrumentation. This is a process of modifying the program's binary file by inserting special calls to produce traces, which can give aid to the developer in finding problems. Dynamic interactive analysis tools capture the execution of the program and provide the user to go forward (and backwards) in the execution time.

A problem reproducer is input data of a regression in an existing workflow. It proves that the workflow used to work in the past and does not work in the present and there was no test coverage for it. A minimal standalone problem reproducer is the minimal version of the problem reproducer running on various system platforms. The minimal standalone problem reproducer usually visits the minimal execution path in the system with minimal amount of data triggering the regression. In essence, this mechanically eliminates the common execution paths in the past (reference) version of the system and the regressed version of the system.

Differential debugging is performing analysis of the regressed system and identifying the cause of the unexpected behavior by comparing to a previous version of the same system. The programmer is offered a set of tools to inspect the internal states and possibly reduce input data to a minimal standalone problem reproducer in order to outline the problematic execution path. The performance differential debugging is a type of debugging targeted at identifying regressions in the efficient utilization of the underlying hardware. The delta debugging is the process of reducing the size of a standalone problem reproducer to a minimum [3].

In this paper we explore a technique to ignore irrelevant to the regression execution paths between a reference and regressed software systems, which we call Interactive Differential Debugging (IDD). Our debugging infrastructure aims to also provide syntactic and semantic tools to compare two versions of the same software and outline functional or performance regressions. The article is divided into six sections. Section 2, Background, describes the related works. Section 3, Architecture and Algorithm, describes an algorithm in pseudo code showing the high-level design and outlines a few code representations and classifies various categories of structural differences. Section 4, Implementation, outlines the architecture of the differential debugging prototype idd. Section 5, Experimental Results, illustrates a few examples where idd performs well, and Section 6 is conclusion.

2. Background

The static analysis includes methods that scan the source code without executing it [4]. For example, the programmer can identify the revision after the regression is introduced and use a tool for visualizing the differences in the source code correlated with the revision, which introduced them. Typically, this method is relatively simple to be performed and gives reliable results. In order to narrow down the regression

revision interval, a binary search known as bisection can be performed to yield results quicker and without manual intervention.

Dynamic analysis of the executables involves their execution and performing instrumentation to inspect their internal states while they are running [5], which in computing is known as type introspection. Debugging while running provides more flexibility to the developer. Unlike static analysis or binary instrumentation, the developer might execute the application line by line and inspect the internal and external state at any given point in the execution time. The latest advancements in some interactive debuggers allow executing line by line forward or backwards.

Widely known dynamic interactive debugging tools are gdb and lldb. They are the debuggers for programming languages like Ada, C, C^{++}, Objective-C, Fortran, Go, Pascal, Modula-2. They provide interactive execution environments, which enable step-by-step debugging. Both debuggers implement user-facing API allowing external tools to control and extend their functionality.

The interactive debuggers are most useful when the software systems are compiled with debug symbols. Software systems compiled with no debug symbols, significantly limit the interactive debugging capabilities, as there is no way to correlate the execution information to the high-level source code. Interactive debugging becomes particularly challenging when the system is compiled in optimized mode. The compiler optimizer is allowed to modify significantly the execution described in the high-level language by shuffling code around, splitting or combining functions.

The printf-style debugging of binaries compiled in optimized mode is often preferred. There are a number of generalized printf-style debugging frameworks. Their idea is to insert calls to a logging facility at key places in the execution, which will produce information about interprocess events [6]. The printed information about execution is often called trace describing its indirect and non-interactive nature. A tracing system can be added directly to the source code, which is preferable, or it can be injected with limited capabilities in the binary using binary instrumentation techniques. Valgrind is an example for a tool capable of injecting analysis code into the binary code of a user program at run time without the need of recompilation, which enables total code coverage of the user program [7].

The trace information about external (to the running process) communication with the underlying operating system and hardware is usually provided by dedicated kernel facilities. The traces are usually used to show differences in applications running in release mode. One of the widely used process-kernel communication tracers is strace. This is a tool for monitoring system calls, changes in the process state and signal deliveries. Perf is a process-hardware communication tracer. It can instrument CPU performance counters, tracepoints, kprobes, and uprobes (dynamic tracing). It is capable of lightweight profiling. Performance counters are CPU hardware registers that count hardware events such as instructions executed, cache-misses suffered, or branches mispredicted. They form a basis for profiling applications to trace dynamic control flow and identify performance regressions.

A limitation by design of the above mentioned systems is they are oriented towards the inspection of a single process (or a process tree) and cannot be used easily

to compare against a reference program or programs. In other words, they cannot debug more than one process per instance and analyze differences against a reference process.

There are few differential debugging tools such as RADAR. RADAR uses static and dynamic analysis to automatically detect feature regressions [8]. The source code of both the base and regressed executables is required in order to generate a report that will be used to generate monitoring scripts. The scripts are used to identify the modified program constructs and put instrumentation algorithms to monitor only the selected functions, their callers and callees. RADAR records the variables values and the control flow of the statements and generates a report based on the traces in the executables after the instrumentation is complete.

VART utilizes regression testing to discover regression bugs not identified by the application test suite [9]. In addition to the base and regressed executable, the tool uses their test suites for the validation analysis. Bounded model checking and dynamic invariant detection is used to automatically detect regressions. The analysis could be divided in two steps – the first uses the base executable and its test suite to generate a set of verified properties. The other performs the regression identification in the second executable using the verified properties and produces related regressive executions that could be used by the programmer to fix the regressions and update the regression test suite.

The downsides of RADAR and VART are they are limited to analyzing only C/C^{++} feature regressed applications. They do not provide interactive mechanisms for the developer to control the execution of the programs and perform type introspection. They are not designed to detect performance regressions.

DPDebugger is a tool for performing automatic performance differential analysis between different inputs used in same executable [10]. The result of the analysis is a report explaining the differences in the execution times of both the base and regression causing inputs in the executables in terms of tracking the control flow graph – which functions were called and how many times they were called. The system produces two types of diagrams – one showing the performance clusters and second showing the control flow information, using machine learning techniques to classify the input data. Then it is up to the developer to read the diagrams and perform debugging if required.

DBDB is a tool for performing differential analysis for identifying diverging behaviors of equivalent interactive debuggers [11]. Internally, the debuggers are represented as models based on Finite State Transducers (FST) in order to abstract out the complex details of each debugger and to generalize DBDB to operate with different types of debuggers. FST is generally a finite state machine but with two tapes and DBDB uses the first tape to take input (user debugging actions) and the second one – to produce output (the result of the debugger). There are three states – not running, running and paused. When transitioning between running and paused states an output is generated by the FSTs and the output difference is a sign for a debugger behavior divergence.

A better differential debugging tool should orchestrate a set of simpler debugging tools providing execution and code modification information. Exposing

enough information to the comparison algorithms enables more precisely to ignore potentially redundant information. Thus, the developer' attention can be focused towards the differences which might have caused the regression. It enables new user interface features such as "divergence" breakpoints. For example, activate the breakpoint when: the parameters of a function call differ from the reference program equivalent function; the object allocation pattern differs; or cache misses differ by orders of magnitude.

3. Architecture and algorithm

The interactive differential debugging system should orchestrate the external and internal analysis of two versions of a program. The implementation should work according to the algorithm:

procedure interactive_differential_debugger(P_1, IP_1, P_2, IP_2)
  while $\exists\langle SP_1, OP_1\rangle \leftarrow P_1(IP_1) \vee \exists\langle SP_2, OP_2\rangle \leftarrow P_2(IP_2)$ **do**
    foreach $i \in 0...N$ **do**
      if $\mu_i(D_iP_1(\langle SP_1, OP_1\rangle), D_iP_2(\langle SP_2, OP_2\rangle)) > \varepsilon$ **then**
        Visualize($\Delta(\langle SP_1, OP_1\rangle, \langle SP_2, OP_2\rangle)$)
      $E_i \leftarrow E(D_iP_1(\langle SP_1, OP_1\rangle), D_iP_2(\langle SP_2, OP_2\rangle))$
      if $E_i \neq \varnothing$ **then**
        Notify(E_i, $D_iP_1(\langle SP_1, OP_1\rangle)$, $D_iP_2(\langle SP_2, OP_2\rangle)$)
end procedure

Listing. 1. Interactive Debugging Differential Algorithm

Let P_1 and P_2 be two versions of a program. Let IP_k and OP_k be the input and output data of P_k, $k\in\{1, 2\}$. Let TP_k be the transition of IP_k to OP_k (denoted $IP_k \rightarrow OP_k$), $k\in\{1, 2\}$. Let SP_k be the set of internal program states which transition $IP_k \rightarrow OP_k$, $k\in\{1, 2\}$; μ_i is weighted diff function, where weight depends on the domain D_i; ε is the domain-specific tolerance, which controls whether the change should be visualized. Δ is difference data function. E_i is a list of divergence events. N is number of inspection tools. $D_1P_1...D_NP_1$ are the inspection tools for P_1 which give information about $\langle SP_1, OP_1\rangle$ and $\langle SP_2, OP_2\rangle$, $D_1P_2...D_NP_2$ are the same set of introspection tools for P_2.

Note 1. $D_NP_1 = D_NP_2 \Leftrightarrow IP_1 = IP_2 \wedge OP_1 = OP_2 \wedge (SP_1 = SP_2) \vee (SP_1 \approx SP_2)$;

$SP_1 \approx SP_2 \Leftrightarrow SP_1 \neq SP_2 \wedge SP_1 \triangleq SP_2$, where $\triangleq$ is user-defined predicate.

Note 2. In those terms we can define a regression as when $IP_1 = IP_2 \wedge TP_1 \neq TP_2 \wedge OP_1 \neq OP_2$ and a reproducer the set of SP_2 where $OP_1 \neq OP_2$ for some finite IP_2. Minimal reproducer is min SP_1 and min SP_2 where $OP_1 \neq OP_2$. Usually achieved by minimizing IP_2 step by step.

Note 3. A tool is time-dependent as we ask for information at given execution time. For trace-based tools it is the point defined by the call site. For an interactive tool it is the point where the execution is stopped and state is inspected.

The differential debugging platform should contain a set of tools, which provide information about the execution of the two systems. It should contain a set of mechanisms to filter out the common and uninteresting information and outline the potential differences caused taking a different execution paths. It should be extensible – users should be able to easily plug in another source of execution information and it should allow specialization to the specific debugging workflow, programmer's habits and problem domain. We propose an architecture conforming to the above-mentioned rules on Fig. 1.

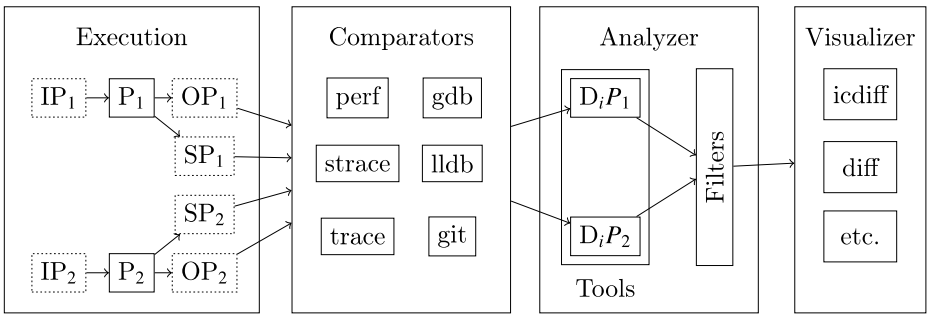

Fig. 1. IDD architecture

There are numerous methods for identification where the control flow or data flow diverges. They perform static and dynamic program execution analysis. Often their efficiency for identifying the cause of the regression is dependent on the type of regression. Modeling the control and data flow happens at multiple abstraction levels. Originally, the programmer develops the system in a high-level programming language (such as C/C^{++}), which is then lowered to machine code in multiple steps. In order for the debugging system to correlate the execution divergence to the source code, multiple execution representations may be necessary. In addition, the representations may simplify writing the necessary comparison tool.

The format of the compiled executable program is machine assembly code. Naturally, the execution information comes in this format but it is augmented with annotations to map the source code (in case debug information is available). The comparisons are done in those two levels by construction. In order to offer different perspective, more models of the execution flows can be constructed. For instance, the assembly code can be disassembled into target-independent intermediate language or in a form of abstract syntax tree [12]. The diverse set of models offer more tools to reason about the execution of a program and it is more likely to understand what is the exact cause of the regression.

Widely used execution flow representations are the control flow graph [13] and the call graph [14]. Comparing the call stacks we implicitly compare the call graphs of the two programs. Comparing the internal state such as local and global variables implicitly reasons about possible changes to the control flow of the program. The

control flow graph and the call graph can be made more explicit and easier to visualize in order to help the programmer discover the issue. In addition, having access to the graphs enables setting conditional breakpoints at the point of divergence of the two programs.

Depending on the problem domain, different representations may be necessary. In cases where we track regressions in the I/O, we need to work with a representation modeling the file system accesses of the underlying file system. The differential debugger should be able to work with the output of for instance *strace* while allowing the user to tune the regression comparators to filter irrelevant information. If we track regressions with respect to image quality, we may use domain-specific comparator such as *imagediff*.

A straightforward starting point is syntactic comparisons. We know the source code differences by construction, as we know the time interval when it stopped working. When this interval is broad, we can use the source code tracking system to narrow it down.

When the execution starts we should apply domain-specific, pseudo-semantic comparisons. This is required because we are running both executables in the same time and the two processes cannot occupy the same address space. We should take this into account and remove this irrelevant information. For example, when we compare certain stack traces we do not want to compare syntactically the addresses of the functions but their call patterns. Similarly, when we compare the allocation patterns we cannot syntactically compare the addresses of the allocated objects but their size and their offsets with respect to some precomputed base address. Filtering this information can happen as a pass before the syntactic comparison. Each execution representation should contain a generic filtering mechanism for comparing two simultaneously running executables and also a handle to the users to extend them with domain-specific knowledge adapting them further to their problem space.

The comparison operations should be customizable where the programmer is given a handle to decide what to filter out. It makes a significant difference in the importance of the information if the programmer debugs an application software system or a microcontroller, for example. In the latter case the content of the CPU registers is much more important than in the former case [15]. When using the performance differential debugging mode the weight information tracking kernel calls is much higher if the system is I/O bound for example.

Some structural differences are expected and unrelated to the regression. They can be classified in several categories:

- Evolutionary – differences which naturally happen due to development. They can be severe when the two versions of the systems are far apart. The issue can be addressed by implementing a different execution representation, which models the operational semantics of the programs. This is outside of the scope of this paper. A practical approach to reduce such effects is to reduce the interval with tools such as *delta* or *creduce* [16].
- Systematic – the interactive differential debugging presumes the two programs being run at the same time. The two processes are loaded in two different address spaces, which makes tracking allocation problems not easy. Fortunately, it is

easy to find such systematic effects on the various comparison kinds if we load the same version of the system twice.

- Deterministic – the operating systems do not guarantee the same process to occupy the same memory space and in fact, they try to randomize it for security reasons [17]. Memory allocation within the process is also implementation-dependent and may not guarantee the same allocation patterns.
- Non-deterministic – differences caused by scheduling, parallelism or severe bugs such as overriding program's stack and memory leaks.

This paper focuses on understanding and controlling simple evolutionary, systematic and deterministic structural differences between the programs.

4. Implementation

Our prototype system, called IDD, implements interactive, dynamic debugging facilities supporting the discovery of functional and performance regressions using generic and domain-specific execution comparison tools. The comparators include syntactic and pseudo-semantic comparisons between control flow and data flow. The comparison uses a reference (baseline) version compared to a regressed version of the same system. The prototype orchestrates a set of debugging tools. It runs the two versions of the program through the same set of debugging tools and in exactly the same order.

The behavior of IDD depends on the configuration loaded. The configuration files describe the active comparators and controls the comparison process. Each configuration file contains hooks for tuning the execution environment prior performing the debugging. The debug control is managed by commands issued to the externally loaded debuggers and the feedback generated is stored for analysis.

The interactive differential debugger multiplexes the user commands to the underlying instances of lldb or gdb. We use tmux (terminal multiplexer) aiding the distribution of multiple active programs in a single terminal window. The output while debugging is situated in a side-by-side split window area called panes. Using an additional controller pane enables the developer to execute debugger commands in both the debugger panes simultaneously and display comparison data in additional helper panes if requested by issuing a user defined command.

Fig. 2 illustrates the terminal-based user interface. The example is trivial – we have two versions of the same program where the initialization of the local variable var1 and var2 is inverted. The top pane is the controller pane and takes input from the programmer. The input is multiplexed simultaneously to the two instances of lldb loaded in the bottom-most panes (lldb 1 and lldb 2). Each of the lldb panes could receive individual commands when it is directly typed in the corresponding panes. The comparison of the stack traces and local variables is shown in the panes.

On Fig. 2 is easy to notice that the comparisons are done on syntactic level and contain some irrelevant information. Our system implements a configuration system that allows developers to specify regular expressions to filter out such syntax meaningless (for the particular debugging use case) differences.

Pre- and post-comparison hooks are available to specialize the environment as close as possible for the particular use.

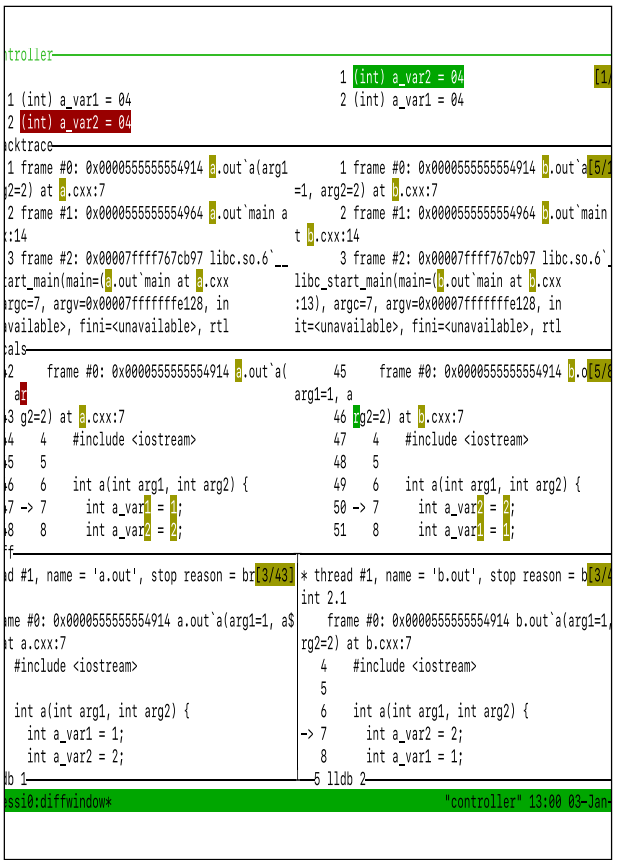

Fig. 2. IDD user interface

The prototype IDD is extensible and supports plugging in more tools producing trace information. It also listens for the output generated by them and is capable to provide differential reports for equivalent output generating applications. The system integrates with perf and results are shown in the next section Experimental results.

We compare the two executables in terms of their control flow and data flow. Comparing the data flow is straightforward – if data diverges then the Analyzer of the idd must issue a warning. The more interesting part is within the comparison of structures part of the control flow.

We propose a mechanism for classification of syntax and data false positives via pluggable functionality called filters. The filters accepts input from both the analysed executables, classifies the detected differences and then signals the developer to decide whether this is something it should be further inspected or the diff could be ignored as not relevant to the regression analysed. A user decision is always required because it is impossible to prove semantic equivalence between multiple code fragments.

4.1. Systematic structural differences

Programming languages allocate memory using dedicated allocator facilities. The allocator reserves memory starting from an address with certain size and returns a pointer to the beginning of the allocated area. Naturally, when the two programs run at the same time, the allocator returns different pointers to different process address space. This systematic effect can be handled by IDD, its child processes can share information about the currently occupied address space. Allocators are not guaranteed to be deterministic; however, most of them are, unless threading is involved. We can subtract the base process offsets from each pointer to make them semantically comparable. In addition, the effect can be mitigated by replacing the allocator by an IDD-controlled one.

4.2. Evolutionary structural differences

If a variable is renamed in the new version of the program, it will be detected as a difference on the source level; however, the rest of the comparison representation will provide enough information to detect if it denotes the same memory address. In order to classify the different names of the two variables analysed, IDD would compare their values and would consider it a false positive. Then, it would raise a warning to signal to the developer that a possible variable name change is performed between the revisions.

When the positions of the local variables declarations are changed this would cause a difference in the source code. However, this kind of change typically should not create a semantic change in the application behavior unless the initialization causes side effects. Internally, IDD creates a list of the local variables for each instance, sorts it and performs the comparison. Detecting a change in the order of declaration should be classified as a suggestion for the developer to inspect.

It is not uncommon to modify the sequence of the arguments in subroutine. If they are of different type a syntactic error would be generated by the compiler. A semantic problem might arise when the variables are of the same type. Then the compiler would not issue an error because the subroutine signature would remain the same. The call to the refactored subroutine would be a valid syntactically but it would result into semantic error.

5. Experimental results

Although the prototype is at an early stage of development, we experimented on real world codebases. We performed regression analysis on the clang compiler. We did performance regression analysis in the high-energy physics data analysis package ROOT [18]. In addition, we show how to detect regressions in the fragile automatic vectorization of in an example program.

5.1. Regression analysis

PR43674 [19] reports a recent regression in the clang compiler constexpr interpreter accompanied with a problem reproducer. Unfortunately, the reproducer expands

around 30K lines of non-trivial code. The reject-valid nature of the problem makes it very difficult to reduce via delta debugging.

Running IDD and inserting a breakpoint in the EvalateSwitch function in the constexpr interpreter is able to locate the regression by inspecting the state changes as it could be seen on Fig. 3. Retrospectively, the regression was introduced in revision r372538 and was fixed in r374954. We ran IDD on r372534 and r374952 respectively. Even though the two versions have evolved over the last 2418 revisions, IDD saved significant amount of time in localizing the issue. It can offer quicker convergence if we had ran within closer intervals. Those kinds of intervals would allow state comparisons that could point the difference almost automatically.

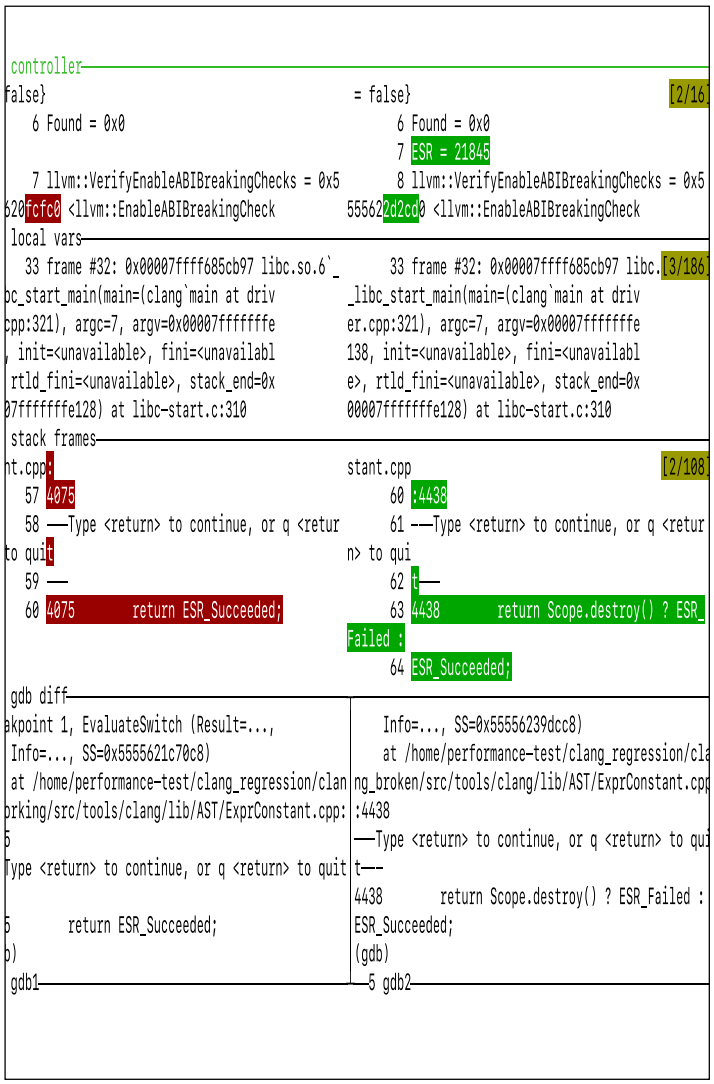

Fig. 3. Clang regression isolated in IDD

The discovered bug originates in the failure of clang to evaluate constants in switch statements. When the scope successfully destroys itself, it should return ESR_Succeeded rather than ESR_Failed. The actions performed for the bug identifications were loading the reference and regressed executables for side-by-side debugging in IDD. Then line-by-line execution was performed via the controller pane. Comparing the internal states of the two executables made the differences trivially observable. As a result, the identification of the point of divergence in the ExprConstant.cpp file was easy to identify.

5.2. Performance analysis

The integration with strace is demonstrated in Fig. 4. We downloaded ROOT v6.18 from the website and compiled ROOT master as of 02.01.2020. We instrumented the expensive system call stat for the initialization and deinitialization of root.exe (strace -e stat root.exe -l -b -q). Then we apply a set of custom filters to remove syntactic differences that are semantically equivalent. For example, the base directory and pointers, which are irrelevant for the investigation. There we noticed there were calls to stat that return ENOENT (No such file or directory). ROOT v6.18 has 251 versus 3522 for the ROOT master. We notice two issues:

- Installed ROOT does a start on the folder that it was build in which by definition does not exist when it is installed.
- ROOT master does many stat calls for virtual files that do not exist on disk.

Fig. 4. Integration with strace

Both issues were reported for clarification [20][21]. In future, we can connect this static information to the interactive debugging facilities and show more information about the execution such as call stack.

The integration with linux perf is demonstrated in Fig. 5. The performance degradation of the updated sample is outlined in pane cache utilization. We instrument a program iterating over a two dimensional array. Program a.out processes data by row while program b.out processes data by column violating the spatial locality of the CPU L1 cache leading to the performance regression.

Fig. 5. Integration with perf

6. Conclusion

We presented a prototype with a flexible architecture capable of conducting interactive differential debugging. We demonstrated how IDD could help discovering regressions in large software systems such as the clang compiler. We also showed how we could compare information from other external sources such as perf to discover performance regressions.

Conceptually, domain-specific comparators are supported to analyze different types of applications and their intermediate results. Interesting domains for our research could be images, databases, ray tracing intermediate results. Each of those domains forms separate domains with its own domain specific aspects and could largely benefit from the differential debugger concept.

The future plans of IDD include the implementation of more software layers to support more external comparators. Major part of the performance regressions emerge also as elevated power on memory consumption. IDD works with different code representations such as control flow graphs and call graphs implicitly. Depending on the user domain, these structures can be generated explicitly. New representations to handle images ought to be developed. Interesting research area is the generation of minimal problem reproducers based on the diverged code paths. Transforming the intermediary data back to input data reproducing the required CFG greatly aid the use cases where delta debugging is essential.

References

1. Wong, W. E., R. Gao, Y. Li, R. Abreu, F. Wotawa. A Survey on Software Fault Localization. – IEEE Transactions on Software Engineering, Vol. **42**, 2016, No 8, pp. 707-740.
2. Canfora, G., A. Cimitile. Software Maintenance. – In: S. K. Chang, Ed. Handbook of Software Engineering and Knowledge Engineering. World Scientific Publishing Company, USA, 2001, p. 940.
3. Collofello, J., S. Woodfield. Evaluating the Effectiveness of Reliability-Assurance Techniques. – Journal of Systems and Software, Vol. **9**, 1989, No 3, pp. 191-195.
4. Jackson, D., M. Rinard. Software Analysis: A Roadmap. – In: A. Finkelstein, Ed. Proc. of Conference on the Future of Software Engineering, ACM, USA, NY, 2000, pp. 133-145.
5. Ball, T. The Concept of Dynamic Analysis. – In: O. Nierstrasz, M. Lemoine, Eds. Software Engineering – ESEC/FSE'99, Springer, Berlin, 1999, pp. 216-234.
6. Edwards, T., J. Charles. Patent US6539501 (Method, System, and Program for Logging Statements to Monitor Execution of a Program), 2003.
7. Nethercote, N., J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. – SIGPLAN Not., Vol. **42**, 2007, No 6, pp. 89-100.
8. Pastore, F., L. Mariani, A. Goffi. RADAR: A Tool for Debugging Regression Problems in C/C^{++} Software. – In: Proc. of 35th International Conference on Software Engineering (ICSE), IEEE, USA, San Francisco, 2013, pp. 1335-1338.
9. Pastore, F., et al. Verification-Aided Regression Testing. – In: Proc. of International Symposium on Software Testing and Analysis, ACM, USA, San Jose, 2014, pp. 37-38.
10. Tizpaz-Niari, S., P. Cerny, B. Y. Chang, A. Trivedi. Differential Performance Debugging with Discriminant Regression Trees. – In: 32nd AAAI Conference on Artificial Inteligence (AAAI'18), 2018, pp. 2468-2475.
11. Lehmann, D., M. Pradel. Feedback-Directed Differential Testing of Interactive Debuggers. – In: Proc. of 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, USA, NY, 2018, pp. 610-620.
12. Vassilev, V., M. Vassilev, P. Petrova. SolidReflector: A Multistage, Interactive Decompilation Framework. – In: Proc. of From DeLC to Velspace, Third Millennium Media Publications, United Kingdon, London, 2014, pp. 49-58.
13. Cytron, R., J. Ferrante, B. Rosen, M. Wegman, K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. – ACM Transactions on Programming Languages and Systems, Vol. **13**, 1991, No 4, pp. 451-490.
14. Hall, M., K. Kennedy. Efficient Call Graph Analysis. – ACM Letters on Programming Languages and Systems (LOPLAS), Vol. **1**, 1992, No 3, pp. 227-242.
15. Ireton, M. A., G. Champagne, C. A. Marler. Patent US5901225 (System and Method for Performing Software Patches in Embedded Systems), 1999.
16. Regehr, J., Y. Chen, P. Cuoq, E. Eide, C. Ellison, X. Yang. Test-Case Reduction for C Compiler Bugs. – SIGPLAN Not., Vol. **47**, 2012, No 6, pp. 335-346.

17. E r l i n g s s o n, U., Y. Y o u n a n, F. P i e s s e n s. Low-Level Software Security by Example, Springer, Germany, Berlin, 2010, pp. 633-658.
18. B r u n, R., F. R a d e m a k e r s. ROOT – An Object Oriented Data Analysis Framework. – Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, Vol. **389**, 1997, No 1, pp. 81-86.
19. Regression in the Constant Evaluation of "Switch" Statements. **https://llvm.org/PR43674.**
20. Installed ROOT Makes Stats to Its Jenkins Build Folder. **https://sft.its.cern.ch/jira/browse/ROOT-10497**
21. ROOT Master Does Redundant Stats on Virtual Files. **https://sft.its.cern.ch/jira/browse/ROOT-10496**

Received: 24.11.2019; Second Version: 08.01.2020; Accepted: 17.01.2020 (fast track)