Software Debugging and Correction Automation

by Jonathan Ebie

With the advent of increasingly complex software, software debugging and correction has become crucial when errors occur to reduce the time and resources spent finding and fixing the error. In the case of critical systems such as military software designed to protect our country from long range missiles, any errors need to be found and fixed as soon as possible so that the service provided by the critical system can be restored. In addition, hospital software that provides critical information used to provide care for patients must have as little downtime as possible when errors occur. Making error correction and debugging automatic would greatly reduce the time and resources lost finding and fixing these runtime errors that occur after any software deployment.

Automatically debugging and correction errors may initially seem easy to do since all that is required is to find the source of the error and correct the error by adding or removing any necessary programming code. However, the difficulties start revealing themselves when trying to even locate the actual source of the error. Even when the actual source of the error is found, identifying the type of error raises difficulties due to having to carefully test the error-bearing code. Once the type of error is identified, figuring out what correction should be made involves choosing and creating the appropriate solution out of all the possible solutions. Finally, the solution to fix the error by modifying the code has be implemented.

Interestingly, no one way to debug has been developed which results in many different methods to locate the error and determine what type of error it is. All of these debugging methods rely on being able to replay the past processes to find the error with the result that an error report is generated for the developer of the software. By involving error reports and developers, precious time and resources are spent waiting for the error report to get to the developers so that the developers can determine and fix the error. To avoid this waste of time and resources, the complete debugging and correction process should be automated.

To this end, I propose that several different methods of debugging are combined into one complete software package with an option of which debugging method to use. These methods would use various means of actively storing vital data and depending on the method, replaying using the stored data to find the error. The culmination would be determining the actual location of the error and determining an appropriate fix that would then be implemented by modifying the code.

Accordingly, combining ideas from Flashback, ReCrash, BugNet, and Darwin, the error for many different types of programs including object-oriented programs and up-to-date software versions can be found and isolated. Using the data already gained by one or more of the given methods will then allow the correction part of the software to identify what the actual error has occurred by backtracking till the cause for first error in the error chain is found. This could be the first error or any errors if any before the reported error. After the actual error is found and identified, one of the solutions in the solution library

would be implemented to modify the code to fix the error. Finally, a report would be generated for the convenance of the developer of the debugged software using Clarify, a system for improving error message readability.

Starting with Flashback, this debugging technique enables accurate debugging by rolling back and replaying code from previously saved checkpoints called shadow processes. To enable accurate replay, all interactions with the system are logged so that deterministic replay is achieved. By making Flashback a light-weight operating system extension, little overhead occurs making quick rollback and replay of shadow processes possible. Flashback was developed to aid in debugging software in which time becomes critical to finding the actual cause of the error. Time becomes critical if several hours or days have to pass before the error occurs and when any such time-critical error occurs, completely restarting the program may not be plausible or advisable due to wasted time and resources.

Before rollback and replay can be executed from any previous shadow process, a shadow process or a passive snapshot of the currently running process has to be created. To do this, the *Checkpoint()* method is called to capture the current state of the running process with a unique *StateHandle* returned to be used for rolling back to the saved state if necessary. Process memory, registers, file descriptor tables and any other relevant data are examples of what be saved about the current state when *Checkpoint()* was called. The system call *Fork()* is used create the shadow process which is then stored in main memory and immediately suspended. Although the shadow process or current snapshot

has been saved, interactions between the system and the process may change requiring that all system interactions be logged. This requirement is necessary to have deterministic replay so that the error can be accurately reproduced. Finally, checkpoint calls can be done automatically or explicitly controlled by the programmer. Automating the calling of *Checkpoint()* add convenance with no additional change to the source code however, having the programmer decide when *Checkpoint()* is called enables more appropriate and effective checkpoint calls. If at anytime a shadow process is not required anymore, *Discard()* is called which permanently discards the specified shadow process.

Before rollback for single and multi-threaded is discussed, two more aspects about shadow processes should be noted. First, two or more shadow processes can be stored at the same time to allow thorough debugging in case the latest shadow process does not produce the actual error. Secondly, the shadow process is actually written using a method called write-on-copy to reduce overhead by only saving the current process state into read-only virtual memory after the first write to any page. This writing methods is activated after *Checkpoint()* is called.

Continuing to the actual rollback methods, single thread processes first reloads the latest shadow process saved with previously saved shadow processes reloaded only if the error was not reproduced. When rolling back multi-threaded processes, the complete process state is recorded so that when rollback occurs, the current state on all of the processors are consistent to allow accurate replay. Also processor synchronization is reset to the reloaded state.

After rollback has occurred, replay then is possible. When *Replay()* is called, the process starts executing from the reloaded shadow process with the assurance that the interactions used for the replay will be the same as the original interactions that had occurred between the current process and the operation system. This assurance is due to the *log* mode in Flashback which is set when *Checkpoint()* is called. *Log* mode enables the return value and side effects of a system call to be logged so that when *replay* mode is activated after *Replay()* is called, the same return values and side effects occur exactly as when the current code had run. Examples of these systems would be filesystem, virtual memory, network related, process controlled, interprocess communication, and utility functions.

With the second debugging method ReCrash, object oriented software can be easily debugged due to ReCrash using objects which would be the individual methods themselves to determine which values to store on the generated shadow stack. This debugging process is split into two parts, the monitoring process and the test process. The monitoring process monitors the values which are referring to the arguments of the called function and the name of the calling function which are placed in the shadow stack. To make monitoring more efficient, only the methods that are affected are monitored with the option *second chance* increasing the efficiency even more. The testing process generates self-contained testing units with each unit being one of the methods being called on the shadow call stack. These test units are meant to reproduce the error through different means so that the error can be found more efficiently.

When monitoring, a shadow call stack of the current call stack is maintained along with the arguments for each method called on the shadow call stack. The main difference involved with monitoring is the depth that the arguments are copied. Since the overhead associated with ReCrash involves mainly time and space resources, the two main ways considered to reduce this overhead is monitoring fewer methods and copying arguments to lesser depth.

When copying arguments to the shadow stack, four different copying strategies can be used. These are reference, shallow, depth-*i*, and deep copy. Reference only stores the reference of the the argument while shallow stores the actual values of the argument whether a primitive or a reference. Depth will store the arguments at the most to the depth *i* dereferences with fewer dereferences allowed and deep copy stores the whole state which maximizes the likelihood that the error can be reproduced. In addition, the option **used-fields** can be activated with the last three copying strategies to activate even deeper copying on used fields that are read or written to in the method itself. Thus, the overhead will change depending on which copying strategies is chosen so great thought must be taken on what copying strategy is chosen. Another overhead that must be considered is the number of methods that should be monitored however the solution to reduce this overhead is to only monitor methods that are affected.

Before moving on to testing, which optimizations that developers should choose requires discussion. These optimizations are the copying strategies, used-field option, and second chance. The greatest difficulties are in choosing which optimization should be

used which involves heavily on what the developer's purpose is for their software. Overall, the best overall strategies that reproduced the most errors with acceptable performance overhead was shallow depth with used-field option activated.[2]

Continuing to testing, unit tests were generated by calling each method on the shadow call stack with the appropriate arguments so that the method was executed as it originally was. Each unit test was one of the methods on the call stack and only the test units that reproduced the original error are saved with several test units reproducing the error to increase the chance that actual error is found.

BugNet, the third debugger, uses checkpoints to store register file contents for any desired point during the runtime of the software along with the recorded load instruction values after the checkpoint to find errors in software. Adding to the usefulness of BugNet is the ability to the ability to replay an application's execution across context switches and interrupts[3]. This type of software is designed to be used by the developers while testing new software not yet deployed commercially to find as many errors before deployment.

The basic process of debugging revolves around the idea of checkpoint intervals where the initial checkpoint stores the current state of the register with any load instruction values recorded afterward till the next checkpoint interval occurs. Furthermore, the current checkpoint interval could encounter a interrupt or context switch which will then cause a new checkpoint interval to begin. Both the initial checkpoints and

following load instruction values are stored in the *Checkpoint Buffer* and each checkpoint interval associated with a log known as the first-load log. Importantly, not every load instruction value is recorded every time but only the first time a value is encountered and if that value is ever encountered again in a load instruction, that value is not recorded. To improve recording efficiency, a dictionary compression technique has been added.[3] When replaying checkpoint intervals using the first-load logs, the same register and memory values used to replay the interval are the same as the original software execution.

When checkpointing, critical information such as process id, thread id, program counter, register file contents, checkpoint interval identifier, and time stamp. This information is used associate checkpoint intervals to their first-load logs, represent the architectural state at the beginning of the checkpoint interval[3], associate first-load logs with their relevant checkpoint intervals, and show when the checkpoint intervals and their first-load logs were initially created.

Continuing, recording load accesses to memory addresses is only done once on the first memory address access by providing a bit that will be set once initially accessed. Once the bit is set, no more accesses to that memory address will be logged. All of this occurring in level one and level two cache where the first-logged bit is preserved when passing blocks into level one cache or when passing blocks into level two cache. If a block is passed out of level two cache but not into level one cache, the first-log bits are reset and all values in the block will be logged again when accessed. Load values logged are used only when the value for any particular load is found in the log with any non-

recorded load value obtained by simulating memory state during replay.[3] Furthermore, all load outputs are logged and are known when to be used by using the *Full L-Count* which stores the number of load instructions skipped between the current load log record and previous load log record. Furthermore, the value stored in *Full Load-Value* is compressed using the dictionary compression technique. All these recording techniques are meant to reduce the amount stored as much as possible.

Using the dictionary compression technique enables efficient storage of values to six bits instead of thirty-two bits to save as much memory as possible. Before any value to be logged to the first-load log is actually logged, the value is looked up in the *Dictionary Table* which holds all the frequently occurring load values.[3] If there is room in the *Dictionary Table*, the counter for that value is incremented if that value is already in the dictionary otherwise the value is added to the entry that has the lowest count value. The lowest entry is chosen when multiple entries can be filled.

Interrupts and context switching are handled by completely ignoring the output of the loads during the this time. To ensure that state stays consistent, a new checkpoint is created after the end of the interrupt or context switch with all first-load bits reset. External input is also handled using this previous method.

Before replay can accomplished, the first-load log for the current checkpoint is used to initialize the complete state. After all initialization has been completed, the actual execution starts with breaks at every load instruction. Once a load instruction has been reached, the decision has to be made whether to obtain the value from the first-load log or

memory. This is achieved by using the *LC-type* to decode the actual value of the *LC-Count* which is then used to determine whether memory or first-load log is accessed. Finally, if the first-load log is chosen, the *LV-Type* determines whether the full thirty-two bit value is used or the 5 bit value from the *Dictionary Table* is used.

Darwin, the last debugging method and most complete debugging method, determines what errors are caused between two software versions. The main issue that arises when new software versions are deployed is that previous inputs that run successfully in the old software version cause errors in the new software version. To locate the actual source of the error in the new version, the path of the failed input and the path of the successful alternate input are compared to find the differences which would be the error causing the caught error. However, this technique can only be used if the conditions that the two inputs follow the same path in the old version and follow different paths in the new version.[4]

Before the successful alternate path trace can be produced, the alternate input has to be generated. Using the formula $f \wedge f'$[4] where f and f' are path conditions which all take the same path of successful execution. The failing input is used to generate the path conditions which are then used by the previously given formula to find the alternate input. Once the alternate input is calculated, the trace path for the successful run of the new versions is produced and compared with the trace path of the failing run to reveal the actual cause of the error. In the case that $f \wedge f'$ is unsatisfiable by having one of the conditions not being met, the formula $f' \wedge f$ can be used determine the alternate input.

This uses the old software version with the generated alternate taking a separate path than the failing input with the produced traces being compared. Any differences found show the error that is causing the new software to produce the originally caught error.

When any error is caught and successfully reproduced, the true cause of the error can then be found by using the section of software code used to successfully reproduce the originally caught error. Since Flashback and BugNet both use checkpoint strategies, identifying the error will be similar while ReCrash due to using complete methods to duplicate the error will have a different method to identify the error cause. Finally Darwin requires little error identification because the old and new software versions are being compared thus showing any errors which would be the differences between the old and new version of the software.

Due to Flashback and BugNet both using checkpoints to record the complete state associated at any point during the execution of the software, identifying the actual cause of any caught and reproduced error would be same since the software code containing the cause would be obtained in the same manner. The actual task of locating the actual cause of the error involves determining which variable(s) are directly and indirectly involved causing the originally caught error. Once these variables are determined, the software code that reproduced the software error is reloaded and re-executed to find the variable(s) that were changed or not changed in such a manner to ultimately cause the caught error. If more software code is required to obtain the actual cause, previous checkpoints can be loaded with execution starting from the loaded state to ensure that the actual cause will be

found.

Variable(s) directly causing the error are first determined by locating the software code producing the error and recording the variable(s). After that, a search is made for those variable(s) in the software code interval used to reproduce the originally caught error. When any of these directly affecting variable(s) are found, any variable directly affecting these variable(s) are considered to be indirectly causing the error. Any such indirectly causing variable(s) are then recorded as well. When all such variable(s) are recorded, the code interval that reproduced the error is re-loaded and re-executed so that the values of the variable(s) that caused the error and variable location information can be recored. Once the values have been recorded, they can be back traced to see the value(s) that were incorrectly changed. The locations of these variable(s) containing the incorrect value would be where the actual source of the error would be located.

Although it would seem that the same error identification method that works for Flashback and BugNet should work for ReCrash, the debugging process of ReCrash is sufficiently different to require its own error identification method. ReCrash takes advantage of the objects in object-oriented software to debug software. Because of this, ReCrash is limited to object-oriented software however, due to the growing importance and use of object-oriented software, this limitation can be ignored for the most part.

ReCrash considers individual methods as single objects and from these individual methods constructs single unit tests. These unit tests are what is used to reproduce the captured error with the goal of having several unit tests saving different approaches as to

why the error occurred. Again, the variable(s) indirectly and directly causing the error have to be found before searching for the actual cause of the error. This requires that all test units reproducing the original error be executed to see which test units contain variable(s) would cause the error to occur. Once these test units were found, the calculation of these variable(s) must be thoroughly checked with additional variable(s) indirectly causing the error being recorded. This would continue until all such variable(s) were discovered which is crucial to find the actual cause of the error. After all critical variable(s) indirectly and directly causing the error are recorded, one of the test units reproducing the original error is re-executed so that all values associated with these variable(s) when the error is caused is recored. Once this data has been recorded, it can be back traced to find any variable locations where the value is incorrectly calculated changed, or not changed through missing software code.

The final debugging method Darwin involves comparing the old and new versions of software to determine what the error is which would be difference between the two versions. What is meant by difference only talks about differences between code fragments of the complete software, not the whole software itself. By find the differences, the error is essentially found thus negating any need for any error identification.

After the true causes of the error are located with the each location in the software code recorded along with any relevant data such as variable name and type, appropriate methods to modify the code so that the error does not occur could then be considered.

Crucial to correcting the error is what the correct values should be for the variables containing the incorrect values. Once the correct value is known, any software code can then be added ensure that the correct value is obtained and assigned to the variable. This would apply to all of the variable(s) that contain incorrect values due to incorrect changes, calculations, or missing software code.

Overall, the difficulty lies in determining how the code should be modified so that the correct value is assigned to the variable in question. Before any solutions are implemented, the software code surrounding the incorrectly valued variable is searched to determine what is preventing the variable from providing the correct value. From this search, solutions can be generated for each issue preventing the correct value to being assigned to the variable. These solutions are generated by calling functions to create the basic and advanced structures of code and combining these calls into one function call. In the case where several solutions could be used, several solutions are generated with varying degrees of complexity ranging from adequately fixed to thoroughly fixed. The actual solution to be executed is chosen in part by checking the option *complexity-level* set by the user.

Before any solutions can actually be constructed using the method calls from class *CodeCorrection*, a software code interval must be searched around the variable being assigned the incorrect value. This involves searching the lines of code before and after the variable to an appropriate limit to find any code preventing the variable from getting assigned the correct value. If any code issues are found, a function is generated with the

appropriate *CodeCorrection* method calls inside of the function body. Importantly, two types of code issues can occur while searching lines of code before and after the variable containing the incorrect value. The code issue could be either easy to correct or difficult to correct depending on what code requires modification, any additional code required, and what code should be deleted. If the code requires little modification, very little to no additional code and only code that should be deleted, the actual solution can be generated easily and simply. If on the other hand, the code requires major modification, large amounts of additional code and very little code deleted, the solution thus generated can get very complex and involved. If possible, the simple solutions are preferred as much as possible due to ease in actually executing.

Once the surrounding space of code around a incorrectly assigned variable is thoroughly searched and solutions generated, the option *complexity-level* that was set by the user is checked to help determine which complex solution to execute only if no simple solution is available that can correctly

ensure that the variable is assigned the correct value. This is to make certain that simplest solution is used before any complex solutions are.

The CodeCorrection class contains methods to either add or delete code. Among these methods are basic code constructions such as if and else branch constructions, variable initializers, and other basic code needed to control process flow. Also included is methods to search and delete code in the desired code interval. Knowing which methods to call depend on the original error caught and the complexity of the issue preventing the

variable from being assigned the correct value. The original error caught will determine what set of methods should be used with the complexity of the issue determining the number of solutions with each solution calling the appropriate basic methods differently so as to provide several possible ways to correct the actual cause of the error.

For future work, the process of identifying the actual cause of caught errors can be researched to a much greater degree along with how the solutions to any found true causes should be generated. In addition, research on using a machine learning classifier to make unambiguous error reports for developers convenance.

Overall, the debugging portion has been simplified by using previously defined debugging methods Flashback, ReCrash, BugNet, and Darwin. In this section, the various processes used by each debugger is described. Following this is the process at which each debugger will use to find the actual cause of any caught error. After the true cause is found and identified, the solution(s) to correct the error can then be generated using calls to methods contained in the class *CodeCorrection* and one of the solutions executed.

Citations

[1] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. 2004. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (ATEC '04). USENIX Association, Berkeley, CA, USA, 3-3.

[2] Shay Artzi, Sunghun Kim, and Michael D. Ernst. 2008. ReCrash: Making

Software Failures Reproducible by Preserving Object States. In *Proceedings of the 22nd European conference on Object-Oriented Programming* (ECOOP '08), Jan Vitek (Ed.). Springer-Verlag, Berlin, Heidelberg, 542-565. DOI=10.1007/978-3-540-70592-5_23 http://dx.doi.org/10.1007/978-3-540-70592-5_23

[3] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. *SIGARCH Comput. Archit. News* 33, 2 (May 2005), 284-295. DOI=10.1145/1080695.1069994 http://doi.acm.org/10.1145/1080695.1069994

[4] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. 2009. Darwin: an approach for debugging evolving programs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (ESEC/FSE '09). ACM, New York, NY, USA, 33-42. DOI=10.1145/1595696.1595704 http://doi.acm.org/10.1145/1595696.1595704