Anomaly-Based Bug Prediction, Isolation, and Validation: An Automated Approach for Software Debugging

Martin Dimitrov and Huiyang Zhou
School of Electrical Engineering and Computer Science
University of Central Florida, Orlando, FL 32816
{dimitrov,zhou}@cs.ucf.edu.edu

Abstract

Software defects, commonly known as bugs, present a serious challenge for system reliability and dependability. Once a program failure is observed, the debugging activities to locate the defects are typically nontrivial and time consuming. In this paper, we propose a novel automated approach to pin-point the root-causes of software failures.

Our proposed approach consists of three steps. The first step is bug prediction, which leverages the existing work on anomaly-based bug detection as exceptional behavior during program execution has been shown to frequently point to the root cause of a software failure. The second step is bug isolation, which eliminates false-positive bug predictions by checking whether the dynamic forward slices of bug predictions lead to the observed program failure. The last step is bug validation, in which the isolated anomalies are validated by dynamically nullifying their effects and observing if the program still fails. The whole bug prediction, isolation and validation process is fully automated and can be implemented with efficient architectural support. Our experiments with 6 programs and 7 bugs, including a real bug in the gcc 2.95.2 compiler, show that our approach is highly effective at isolating only the relevant anomalies. Compared to state-of-art debugging techniques, our proposed approach pinpoints the defect locations more accurately and presents the user with a much smaller code set to analyze.

Categories and Subject Descriptors C.0 [*Computer Systems Organization*]: Hardware/Software interfaces; D.2.5 [*Software Engineering*]: Testing and Debugging – debugging aids

General Terms Languages, Reliability, Performance

Keywords Automated debugging, Architectural support

ASPLOS'09 March 7–11, 2009, Washington, DC, USA.

1. Introduction

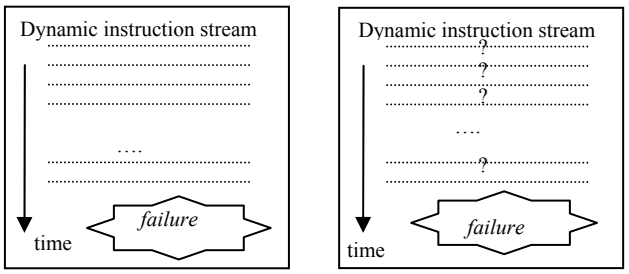
Software defects[1], commonly known as bugs, present a serious challenge for computer system reliability and dependability. Once a program failure such as a program crash, an indefinite loop, or an incorrect output value, is observed, the debugging process begins. Typically, the point of the failure (i.e., the instruction where the failure is manifested) is examined first. Then the programmer reasons backwards along the instruction flow and tries to figure out the cause of the failure. Such backward slicing [1, 14, 33] (i.e., the process of determining all the instructions that have affected the failing instruction) is a tedious and time consuming effort, which may require the programmer to examine a significant portion of the program. Certain bugs, such as memory corruption, make this effort even harder because their effects may manifest only after a very long period of program execution or at unexpected locations. After tracing back the chain of program statements, the programmer creates a hypothesis of what could be the root cause of the failure. He/she then verifies the hypothesis, by modifying the source code and observing whether the failure still occurs. If the failure is still there, then the hypothesis was wrong and the search resumes. To relieve developers of such repetitive exploration, there has been active research toward automated debugging by leveraging the power of modern processors to perform the task.

A key technique used in debugging (automated or not) is backward slicing, which reasons backwards and tracks the origins of a failure. The main issue with this approach is the cost of constructing backward slices, especially dynamic ones. In a recent work, Zhang et. al. [42] proposed an algorithm to significantly reduce the slicing time and the storage requirements so as to make it practical. However, as pointed out in [10, 41], even with efficient backward slicing, a nontrivial portion of the program needs to be examined manually to locate the faulty code.

[1] We use the terminology from the book "Why Programs Fail" [38]. The programmer is responsible for creating a defect in the source code. At runtime, the defect may create an infection in the program state. The infection propagates until it becomes an observable program failure. The terms: software defects, bugs, faulty code and failure root-cause, are used interchangeably.

Another promising technique to facilitate debugging is anomaly-based bug detection [8, 9, 11]. An anomaly detector is either a software or hardware module initially trained to recognize some aspects of correct program behavior during passing program phases/runs (i.e., runs that do not crash or produce faulty results). Then, it is used during a faulty program phase/run to detect violations of the previously learned behavior. Previous works [11, 44] show that such anomalies can lead the programmer to the root cause of hard-to-find, latent bugs. The main issue with those approaches is that they tend to report too many anomalies and it is not clear which anomalies have a cause-effect relation to the program failure.

In this paper, we propose a novel approach to automate the debugging effort and accurately pinpoint the failure root cause. It avoids the expensive backward slicing and overcomes the limitations of the existing anomaly-based bug detection schemes. The proposed approach contains three steps, as illustrated in Figure 1.



(a) Program failure observed

(b) Step 1: Anomaly-based bug/defect prediction

(c) Step 2: Isolating relevant anomalies via dependence

(d) Step 3: Validating isolated bug predictions by nullifying the anomalous effects

Figure 1. Overview of the proposed automated debugging process (the symbol '?' represents a predicted bug).

After a program failure is observed during execution (Figure 1a), the automated debugging process starts. The failure point may be a crash, incorrect results, etc. In the first step, we re-execute the program to reproduce the failure using the existing work on faithful record and replay. At the same time, we enable a set of bug predictors to monitor program execution and signal any abnormal behavior (Figure 1b). In this work, we leverage two previously proposed bug detectors– DIDUCE [11] and AccMon [44], and propose a new loop-count based bug predictor. The combination of various bug predictors offers higher bug coverage as a more complete set of program invariants are monitored (see Section 6).

In step 2, we examine each of the predicted bugs to see whether it leads to the failure and isolate only the relevant ones (Figure 1c). To do so, we construct dynamic forward slices from all the predicted bug points. With the anomaly-based bug predictors, the forward slices include all the instructions that have used anomalous results as source operands, directly or indirectly. If the failing instruction is not in the forward slice of a predicted defect, the predicted defect is considered irrelevant and discarded. Compared to the approaches built upon backward slicing, forward slicing is much easier to compute and can be efficiently constructed in hardware by leveraging tagged architectures proposed for information flow or taint tracking [5, 6, 26, 30].

In step 3, we validate the isolated bugs by nullifying the anomalous execution results (Figure 1d). If the failure disappears, we know that the bug infection chain has been broken, and we have a high confidence that the root cause has been pinpointed. The number of validated defects after this step is very small, even for large software programs like the gcc compiler, showing that the proposed approach accurately pinpoints to the software defect.

In summary, this paper makes the following contributions:

• We propose a novel, automated approach to predict, isolate and validate software defects. The proposed method overcomes the limitations of existing anomaly-based bug detection schemes and avoids the high cost of backward slicing.

• Instead of requiring new hardware, we propose novel ways to reuse existing or previously proposed hardware structures for debugging, thereby reducing the overhead for hardware implementation. We also propose an adaptive partition scheme to overcome hardware resource limitation on bug prediction tables.

• We create a useful software tool[2] using the PIN 2 dynamic binary instrumentation system [19], to emulate the proposed architecture support, which can also be used as a standalone software for automated debugging.

• We perform a detailed evaluation on 6 programs with a total of 7 bugs, including a real bug in the gcc-2.95.2 compiler, which highlights the limitations of existing bug detection techniques. The experimental results show that the proposed approach is highly effective at isolating only the relevant anomalies and pin-pointing the defect location. Compared to a state-of-art debugging technique based on failure-inducing chops [10] our approach locates the defects more accurately and presents the user with a much smaller code set to analyze.

The remainder of the paper is organized as follows. Anomaly-based bug prediction is addressed in Section 2. Bug isolation using dynamic forward slicing is presented in Section 3. Section 4 discusses the validation of the isolated

[2] The tool is available at: http://csl.cs.ucf.edu/debugging.

defect predictions by altering dynamic instruction execution. The experimental methodology is in Section 5 and the experimental results are contained in Section 6. We highlight the limitations of our work and discuss the future directions in Section 7. The related work is presented in Section 8. Finally, Section 9 concludes the paper.

2. Predicting Software Bugs

2.1. Method

Previous research [8, 9, 11] has observed that when infected by a software bug, a program is very likely to behave in some unexpected, abnormal ways. Common abnormal instruction-level behavior includes events such as producing out-of-bound addresses and values, executing unusual control paths, causing page faults, performing redundant computations and possibly many others. Given the correlation between program anomalies and the existence of software defects, several research works [11, 44] have used anomalies to locate the likely root causes of software failures. In our proposed scheme, we use such anomaly detection tools as bug predictors. Anomaly detectors or bug predictors can be viewed as a way to automatically infer program specifications from the passing runs, and then to turn those specifications into 'soft' assertions for the failing run, meaning that we will record the violation of those assertions instead of terminating the program. One attractive feature of instruction-level anomaly detectors is that they usually point to the first consequence of the defect, or the first change from normal to abnormal behavior (i.e., the first infection point). This is very helpful in determining the root cause of latent bugs, such as memory corruption, which may manifest as a failure at an execution point far from the original faulty code. Many bug predictors are possible and they can monitor various program aspects to detect anomalies. Our approach is not restricted to using any particular bug predictor. A combination of multiple bug predictors is preferable as a more complete set of program invariants are monitored. In this paper, we leverage two previously proposed anomaly detectors DIDUCE [11] and AccMon [44] and propose a new one based on loop-count invariance.

DIDUCE exploits invariance in execution results, based on the insight that most instructions tend to produce values within a certain range, e.g. a variable "*i*", is always between 0 and 100. This insight also applies to memory operations, which usually access a certain data structure and produce a limited range of addresses. DIDUCE learns the invariants during passing program runs/phases by initially hypothesizing a strict invariant and then relaxing the invariant as new values are observed. During the failing run, or the bug detection phase, any violation of the learned invariants is reported as a possible cause of the program failure. As a side benefit, DIDUCE can also report "new-code" violations, or instructions, which have no invariance information learned from the training runs. The invariants in DIDUCE are represented in a compact form using a bit-mask, which specifies the bit positions where execution results are expected to vary. The variance between dynamic execution results is computed by a simple XOR operation.

AccMon stands for Access Monitor [44]. Its main idea is that for most memory locations only a few static instructions access a given memory location. This locality is also referred to as the load/store set of a memory location [3]. AccMon exploits this locality to detect memory related defects. After learning the store sets during the training runs, AccMon will make sure that each memory update instruction belongs to the store set of the updated memory location. If it does not, an anomaly will be signaled. Since maintaining the store set for each memory location is expensive, bloom filters are used to test for membership in the set.

In this work, we also propose a new bug predictor to monitor for abnormal number of loop iterations. In general, program defects may result in abnormal control flow behavior and branch mispredictions can used to detect control flow anomalies. One possible approach is to use mispredictions of branches with high confidence, as exploited in [32] for soft-error protection. However, even with a confidence mechanism, the misprediction rate (which is in the range of one misprediction per ten-thousand dynamic instructions) is still exceedingly high for software-bug detection. It is not trivial to reason about the effect that each of those mispredicted conditional branches may have on the observed failure. Therefore, in this paper, we propose to focus on one special type of branches, loop branches. We learn the normal range of loop iterations during passing runs, and detect too few or too many iterations as anomalies during the failing runs. For each anomaly, the instructions in the loop body will be examined for their relevance to the failure using the approach presented in Section 3. We call this bug predictor LoopCount. As we will show in our experimental results (see Section 6), such a simple loop-based bug predictor is effective in catching some interesting memory corruption defects, which DIDUCE misses.

Next, we use a code example from bc-1.06 to illustrate each of the bug predictors. BC is a program from bugbench [16] and is an arbitrary precision calculator language. Figure 2 shows the faulty code in function *more_arrays()*. This function is called when more storage needs to be allocated to an array. It allocates a new, larger array, copies the elements of the old array into the new one, and initializes the remaining entries of the new array to NULL. The defect is on line 18 and is due to the fact that a variable *v_count* is used mistakenly instead of the correct variable *a_count*. Thus, whenever *v_count* happens to be larger than *a_count*, the buffer arrays will be overflown and its size information, which is located right after the buffer, will be lost. This results in a segmentation fault when *more_arrays()* is called one more time, and the buffer with corrupted size information is freed at line 23.

```
1   void more_arrays () {
2     int indx;    int old_count;
3     bc_var_array **old_ary;
4
5     /* Save the old values. */
6     old_count = a_count;
7     old_ary = arrays;
8
9     /* Increment by a fixed amount and allocate. */
10    a_count += STORE_INCR;
11    arrays = (bc_var_array **)
          bc_malloc  (a_count*sizeof(bc_var_array 12*));
12
13    /* Copy the old arrays. */
14    for (indx = 1; indx < old_count; indx++)
15      arrays[indx] = old_ary[indx];
16
17    /* Initialize the new elements. */
      /* defect: incorrect loop condition */
18    for (; indx < v_count; indx++){
        /* infection: overflows its size information */
19      arrays[indx] = NULL;
20
21    /* Free the old elements. */
22    if (old_count != 0){
        /* crash: when the buffer size is corrupted */
23      free (old_ary);
24    }
25  }
```

Figure 2. Incorrect loop condition in *bc-1.06* leads to an overflow in a heap buffer '*arrays*', which corrupts its size information. The subsequent call to *free(old_ary)* causes a segmentation fault due to the corrupted size information.

To detect the bug in Figure 2, we initially trained all the bug predictors using several BC runs such as computing prime numbers, square roots, etc. Then, we executed BC with a specially crafted input program, which was able to trigger the defect and overflow the buffer on line 19. The store instruction in assembly responsible for the overflow is: "movl $0x0, (%eax, %ebx,4)". During the passing runs, DIDUCE has learned the range of addresses that this store instruction accesses. During the failing run, more storage is required and the function *more_arrays()* is called with requests for larger arrays. This causes the loop on line 18 to execute more times than usual and the store instruction on line 19 to access a wider range of memory addresses. DIDUCE detects this abnormal behavior and signals an anomaly. It is interesting to note that if we changed the variable *v_count* to the correct variable *a_count*, DIDUCE would still signal an anomaly, which would then be a false positive. This implies that DIDUCE does not exactly detect the defect, but rather the abnormal behavior triggered by the buggy input. In our particular example, DIDUCE signaled an anomaly on line 19 in Figure 2, which is the immediate infection point of the defect on line 18. Therefore, we consider it a successful detection of the bug. Besides this anomaly, DIDUCE also signaled twenty-three false-positive ones, one of them on line 15. The rest of the false-positive anomalies include eighteen "new-code" and four non new-code anomalies in the same and other functions. AccMon also detects the defect in Figure 2, because the store operation on line 19 does not belong to the store set of the corrupted memory location. In our implementation, AccMon also signaled another 67 false-positive anomalies (3 in this function and 64 in other functions). LoopCount detected the abnormal behavior in the for-loop on line 18, whose loop condition is the defect. It also signaled an anomaly in the for-loop on line 14 (a false positive) and thirty-four additional false positives in other functions.

One issue with AccMon is that virtual addresses of memory objects allocated on the heap or stack may vary among different program runs. Therefore, the invariants obtained during the passing runs will not be useful for the failing run. To solve this problem, AccMon uses a special call-chain naming for stack and heap objects by intercepting each memory allocation. In our implementation, we do not use the call-chain naming strategy since we assume no compiler/system support for intercepting memory allocation. Instead, we use an offset address relative to the current stack pointer for stack accesses. For heap accesses, virtual addresses are used and our experiments show that it results in a higher number of false alarms than reported in [44] but is still effective in detecting relevant anomalies. The reason is that that we report new store addresses as anomalies as well. AccMon utilizes different heuristics and confidence mechanisms to reduce the number of false-positive anomalies. For example, the compiler is used to identify possible array and pointer accesses, which are more likely to contain software defects. Memory accesses, which are not pointer or array references, are not monitored. This optimization reduces false positives but may cause AccMon to miss some bugs. In our implementation, we choose to monitor every memory update and use automated bug isolation to eliminate false positives (see Section 3).

2.2. Architectural Support

The bug predictors described in this section are suitable for hardware implementation. The reason is that modern processors already exploit various program localities to improve performance. Our proposed LoopCount bug predictor is light weight since it can simply reuse existing loop-branch predictors. The invariants used by DIDUCE and AccMon, can be captured using cache structures with limited sizes. In [7, 24], hardware implementations of DIDUCE with limited sizes are proposed for either soft-error detection [24] or software bug detection [7]. In this work, we model DIDUCE as described in [7] by tracking anomalies in store addresses. This approach reduces the pressure on the DIDUCE table and is effective because most bugs manifest through memory operations [16]. A hardware implementation using cache-like structures was proposed for AccMon in [44]. The most frequently accessed addresses and their bloom filters are maintained in a small cache with only 4 to 8 entries. In this paper, we implement the AccMon bug predictor as a table of bloom filters and also assume that part of this table can be cached in hardware.

As highlighted in [44], efficient architectural support for anomaly detectors has the benefits of minor performance

impact, high accuracy in run-time event measurement, and portability. The high performance efficiency also makes it possible for the detectors to be used in production runs to generate detailed error reports. In our proposed automated debugging approach, the hardware implementation also enables efficient ways to change or invalidate dynamic instruction execution – the third step of our approach. For example, if an out-of-range store needs to be skipped during the validation step, once an anomaly detector captures such an out-of-range store address, it can inform the processor to invalidate the corresponding dynamic instance. This way, the validation can be performed without source code modification or binary instrumentation.

A concern, however, lies in limited hardware resource, which may cause both an increased number of false-positives as well as a loss of bug detection coverage due to replacements in prediction tables. To solve this problem, we propose to adaptively partition a program into code regions and use multiple runs to cover the whole program. In each run, only one of the regions is monitored. The policy for determining the code regions is as follows. If the number of replacements in the predictor table exceeds a threshold T, then we split the current program/region into two and monitor each of them separately. We perform this partition recursively, until the number of table replacements becomes less than T. We also use a PC-based XOR function to generate a uniform distribution of instructions among code partitions. In our experiments in Section 6, we use a 2K-entry prediction table and our splitting threshold T is 20. The results show that the performance of this approach is very close to that of a 64K-entries table and all root causes are successfully detected.

3. Isolating Relevant Bug Predictions

3.1. Method

As discussed in Section 2, anomaly-based bug predictors are capable of identifying abnormal behavior, which may be a reason for the program failure. However, two problems remain. First, it is not clear which anomaly(s) points to the actual defect and which ones are false positive. In the example in Figure 2, among the 24, 68 and 38 anomalies detected by DIDUCE, AccMon and LoopCount, respectively, the programmer is expected to go through each of them to evaluate its validity. Depending on the size of the software program and the quality of the training inputs used to train the bug predictors, the number of false positives can become very large. Second, there is always a tradeoff between bug coverage and the number of false-positive anomalies. On one hand, producing too many anomalies places a burden on the programmer. On the other hand, if the predictor is made very conservative and signals only few anomalies using some heuristics or confidence mechanism, some defects may go undetected. Our solution to this problem is to allow each bug predictor to signal anomalies aggressively, thereby increasing the coverage at the cost of false positives. Then, an automated process is devised to isolate relevant anomalies, i.e., those that actually lead to the program failure, instead of placing the burden upon the programmer. To achieve this, we construct dynamic forward slices of each anomaly and retain only those anomalies whose forward slices contain the point of failure. The relevant anomalies can also be extracted from the dynamic backward slice originating from the point of failure. While both approaches are possible, computing dynamic forward slices is much less expensive than computing dynamic backward slices.

3.2. Architectural Support

In this paper, we propose to construct the dynamic forward slices in hardware by leveraging tagged architectures proposed for information flow tracking or taint tracking [5, 6, 30]. In our implementation, each memory word and each register contains a single extra bit, which we call a token. When bug predictors detect an anomaly, they will set the bit (the token) associated with the destination memory location or register of the violating instruction. Subsequent instructions propagate this token based on data dependencies. When the program eventually fails, we examine the point of failure for the token. If the failure point is a single instruction, e.g. causing a segmentation fault, then we examine the source operands of the instruction for the token. If the failure point is a function call, such as a call to output erroneous results, or a call hung in infinite recursion, then we examine the function call parameters for the token. If the token is present, this means that there is a relevant anomaly among those signaled by the bug predictors.

We illustrate this point with the example from Figure 2. The for-loops on lines 14 and 18 iterate more times than usual. The bug predictors signal anomalies and mark with tokens the two store instructions corresponding to: “arrays[indx]=old_ary [indx]” on line 15 (false-positive) and “arrays[indx]= NULL” on line 19 (buffer overflow). Due to the overflow, the memory location, which holds the size information of arrays, is overwritten by “arrays[indx] = NULL”. Therefore, it will be marked with the token. When the statement *chunk_free* inside the function *free(old_ary)* on line 24 crashes the program, it will carry the token because the corrupted size information is used as its parameter.

Since we have only one token and potentially many anomalies, we do not know which specific anomalies are responsible for propagating the token to the point of failure. In this example, only the one corresponding to the statement “arrays[indx] = NULL” on line 20 is responsible for marking the corrupted memory location. To isolate the relevant anomalies, we leverage the delta debugging algorithm proposed by Zeller [38, 40]. The delta debugging algorithm is a divide-and-conquer approach, which is used to automatically simplify and isolate failure inducing input [40], failure inducing differences in program state [4], as well as failure inducing cause-effect chains [37]. Conceptually, our anomaly isolation algorithm works as follows. First we divide the anomalies in half, and allow

an1	**an1**	an1												
an2	**an2**	an2												
an3	**an3**	an3												
an4	**an4**	an4												
an5	**an5**	an5												
an6	**an6**	an6												
an7	**an7**	an7												
an8	**an8**	an8												
an9	**an9**	an9												
an10	**an10**	an10												
an11	**an11**	an11												
an12	**an12**	an12												
an13	an13	**an13**	**an13**	an13	**an13**	an13	an13	an13	**an13**	an13	**an13**	an13		
an14	an14	**an14**	**an14**	an14	**an14**	an14	an14	an14	**an14**	an14	an14	**an14**		
an15	an15	**an15**	**an15**	an15	**an15**	an15	an15	an15	an15	**an15**				
an16	an16	**an16**	**an16**	an16	an16	**an16**	an16	an16						
an17	an17	**an17**	**an17**	an17	an17	**an17**	an17	an17						
an18	an18	**an18**	**an18**	an18	an18	**an18**	an18	an18						
an19	an19	**an19**	an19	**an19**	an19	an19	**an19**	an19						
an20	an20	**an20**	an20	**an20**	an20	an20	**an20**	an20						
an21	an21	**an21**	an21	**an21**	an21	an21	**an21**	an21						
an22	an22	**an22**	an22	**an22**	an22	an22	an22	**an22**					**an22**	an22
an23	an23	**an23**	an23	**an23**	an23	an23	an23	**an23**					**an23**	an23
an24	an24	**an24**	an24	**an24**	an24	an24	an24	**an24**					an24	**an24**
◌		◌	◌	◌	◌			◌	◌		◌	◌		◌

Figure 3. Using delta-debugging to automatically isolate relevant anomalies. The symbol ◌ means that the token is present at the failure point. Anomalies marked in bold are allowed to start tokens while those in grey are not.

only one half to propagate the token. If the selected anomalies do not propagate the token to the failure point, then we discard them and continue the process with the other half. If both halves propagate the token to the failure point, this means that there is at least one relevant anomaly in each half. In this case, we increase the granularity (divide into quarters and eighths, etc) and continue the process. The algorithm terminates, when we cannot divide the anomalies any further and we have discovered all the relevant anomalies. We illustrate the process in Figure 3 for our running example of bc-1.06. We start with the 24 anomalies, detected by DIDUCE. In each run, the anomalies marked in bold in Figure 3 are selected to propagate the token, while the anomalies in grey are ignored. After 15 delta debugging iterations, the anomalies are reduced to only three. The defect "arrays[indx] = NULL" on line 19 is among those three. The other two isolated anomalies are responsible for setting up the parameters to the function call chunk_free, which crashes the program, and thus they are on the defect infection chain. In general, the worst case complexity (i.e., the number of delta debugging runs) is n2 + 3n [40], where n is the number of anomalies. The process for AccMon is identical. For LoopCount we mark all instructions in the loop body with a single token.

4. Validating Bug Predictions

After isolating relevant anomalies, we are typically left with only few remaining bug predictions. Each of these remaining ones forms a hypothesis that it is the root cause of the failure. As addressed in Section 1, the final step of a debugging process is to validate the hypothesis by modifying the suspicious code and observing if the failure disappears. We propose to automate this part of the debugging effort as well. We validate each hypothesis individually, by applying a fix and observing whether the failure still occurs. The fix is simply nullifying (or turning into a no-op) the dynamic instance of the violating instruction to prevent it from updating memory or its destination register. In the case of 'new code' anomalies, we do not know which dynamic instance of the instruction is causing the problem, and thus we nullify every dynamic instance. Consider again our example from bc-1.06. If we do not allow the dynamic instruction: "*arrays[indx] = NULL*", which overflows the buffer, to be executed (i.e. if we turn the instruction into a no-op), the size information will not be corrupted and the segmentation fault disappears. In general, after nullifying a dynamic instruction, four possible outcomes can be expected:

• **Application execution succeeds.** We consider execution to be successful, if the failure symptom (crash, infinite loop, corrupted results) disappears and the output produced by the program is correct. In this case, we say that we have validated a hypothesis, and we have the highest confidence that the selected anomaly points to the defect, or is at least part of the defect infection chain. In bc-1.06, after nullifying the root cause instruction, the program does not crash and prints the correct output to the screen. Such dynamic nullification can also serve as a temporary bug fix, if necessary.

• **Application execution does not crash.** The program does not crash (or hang in infinite loop), but it produces incorrect or missing output. Such outcome is possible when the nullified instruction is vital to the computation of correct results. We can also expect this outcome, when dynamic nullification causes the program to take a different control path or exit prematurely.

• **Application execution fails.** In this case, even after nullifying the violating instruction, the application fails with the same symptoms as before and with the same call-chain stack. This does not necessarily imply that the bug is false positive. The reason is that the failure may be a result

of multiple infections of a single or several defects and fixing one of them is not sufficient to eliminate the problem. Therefore, if after isolation, more than one relevant anomaly remains and nullifying them one-by-one results in the same failure symptoms, we propose to nullify a combination of several dynamic anomalous instructions together. This approach becomes expensive if the number of anomalies is large because of the exponential number of possible combinations. In such a case, we could try to prune the search space by nullifying violating dynamic instructions based on their dependency relationship. For example, all anomalies in the same dependence chain can be nullified at once. Such dependency exploration is left as future work.

• **Application execution outcome is unknown.** In some cases, nullifying a dynamic instance of a violating instruction causes the application to terminate with a different error from the original failure symptom. In this case, we cannot be sure whether the anomaly directly leads to the defect, and we mark it as unknown. In bc-1.06, after nullifying the other two isolated anomalies, the function call parameters to *chunk_free* become incorrect and bc-1.06 crashes with a different error. Therefore, we label those two anomalies as unknown.

Our experimental results show that nullifying the results of violating instructions is a simple, but effective approach to validate the relevance of anomalies. However, this part of our approach is not guaranteed to succeed because of incorrect outputs or unknown execution outcomes. Thus, we use validation to rank the isolated bugs from most to least relevant: execution succeeds, execution does not crash, execution is unknown, and finally execution fails. In the running example of bc-1.06, the root cause is ranked highest since "execution succeeds" when it is nullified.

5. Experimental Methodology

5.1. Dynamic Binary Instrumentation

As a proof of concept and a working debugging tool, we implemented our approach using the Pin 2 dynamic binary instrumentation system [19]. In this software implementation, instrumentation functions are inserted before each dynamic instruction (including instructions in shared libraries). The instrumentation functions perform anomaly detection, token propagation, and selective nullification of dynamic instructions, as described in Sections 2, 3, and 4, respectively. The experiments were conducted on a Red Hat Linux 8.0 system with an Intel Xeon 3.0 GHz processor. Because of the IA-32 instruction set architecture, we wrote custom token propagation rules for certain instructions. For example, in IA-32 it is a common practice to produce 0, by XORing a register with itself, such as 'XOR %eax, %eax'. In this case, we reset the token of the destination register %eax. IA-32 also contains a variety of conditional move instructions, MOVcc. If a certain condition is satisfied the move operation is performed, otherwise the instruction turns into a no-op. For these instructions, we evaluate the condition and propagate the token only if the instruction will actually be executed. Other instructions, such as PUSH and POP, place or retrieve a value from the stack and at the same time increment or decrement the stack pointer. For those instructions, we do not propagate a token to the stack pointer (SP) register. Also, when nullifying the execution results of a dynamic PUSH or POP instruction, we restore the destination memory or register value, but we allow the update to the SP to occur. If we naively removed the whole instruction, the SP would be corrupted and the application would almost certainly crash in an unexpected way.

5.2. Evaluated Applications

Dynamic binary instrumentation allows us to test our approach on unmodified application binaries. We tested our proposed mechanism on six applications and seven bugs as shown in Table 1. Six of the applications are from the BugBench suite [16]. Some of them contain more defects than those shown in Table 1, however we were not able to produce a program failure by exploiting those defects. For example, some memory corruption defects corrupt unused memory regions and do not alter program execution. Although some of these defects were captured by our bug predictors, since no failure can be observed our isolation and validation techniques cannot be applied. The last application that we tested is the *gcc-2.95.2* compiler. The purpose of the *gcc* test is to evaluate the applicability of our approach to large programs. *Gcc* has an order of magnitude more lines of code than any of the BugBench programs. The real bug in *gcc* is analyzed in [37].

6. Bug Detection Results

Summarize your work and discuss future work. Table 2 summarizes the results of our experiments. It reports the number of bug predictions at each stage of our approach: prediction, isolation, and validation (execution succeeds). At each stage, we show the number of bug predictions originating from each type of bug predictors, 'D' for DIDUCE, 'A' for AccMon, and 'L' for LoopCount. The column "Defect Rank" shows where the actual defect ranks among the isolated anomalies, based on the validation step detailed in Section 4 and combining three predictors. In other words, "Defect Rank" represents the *maximum* number of anomalies to be analyzed by the user to locate the actual defect. Taking *gzip* as an example, we have 1 validated (with correct outputs) anomaly from each predictor. Among them, 2 are unique and one of them is the actual defect. So, the rank of the actual defect is reported as 2. In *polymorph*, the actual defect is among the 3 unique isolated anomalies. Although the validation step fails to produce the correct output, the user needs to examine at most 3 anomalies to locate the defect. Results marked as "n/a" mean that the bug corrupted Pin's memory as well causing it to crash.

In our experiments, we compile all the applications with the "-static" option and monitor each instruction, including library code. Without monitoring library code, all bugs

Table 1. Evaluated applications including the defect location and description.

Application	Lines of Code	Defect Location	Defect Description
bc-1.06	17,042	storage.c: 176	Incorrect bounds checking causes heap buffer overflow
gzip-1.2.4	8,163	gzip.c: 1009	Buffer overflow due to misuse of library call *strcpy*
ncompress-4.2.4 2 defects	1,922	compress42.c: 886 and 1740	Buffer overflow due to misuse of library call *strcpy* Incorrect bounds checking causes stack buffer underflow
polymorph-0.4.0	716	polymorph.c: 200	Incorrect bounds checking causes stack buffer overflow
man-1.5h1	4675	man.c:998	Incorrect loop exit condition causes stack buffer overflow
gcc-2.95.2	338,000	combine.c: 4013	Incorrect call to *apply_distributive*_law causes a loop in the RTL tree

Table 2. Bug detection results (The bug predictions are from three predictors: D- DIDUCE, A-AccMon and L-Loop). Applications are compiled with "-static" option and library code is monitored for anomalies.

Application	Initial Bug Predictions			Isolated Bug Predictions			Validated (Application Succeeds)			Defect Rank
	D	A	L	D	A	L	D	A	L	
bc-1.06	24	68	36	3	2	4	1	1	1	**1**
gzip-1.2.4	21	40	19	1	1	1	1	1	1	**2**
ncompress-4.2.4 (*strcpy* defect)	6	7	6	2	2	1	0	1	1	**1**
ncompress-4.2.4 (stack underflow)	2	4	n/a	1	1	0	0	0	0	**1**
polymorph-0.4.0	21	10	20	3	1	0	0	0	0	**3**
man-1.5h1	15	114	46	2	2	0	1	1	0	**1**
gcc-2.95.2	768	1062	666	84	130	47	2	4	3	**9**

except the *strcpy* bugs in *gzip* and *ncompress* can be caught. Monitoring library code, however, slightly increases the initial number of bug predictions, which are quickly filtered by isolation and validation. The results in Table 2 are obtained using large 64K-entry prediction tables. The impact of hardware implementation and limited table sizes is discussed in Section 6.2.

We can make several important observations from the results in Table 2. First, even if a large number of anomalies are signaled initially, they are quickly isolated to only a few. After the validation step, the remaining predictions accurately point to the actual defect. Except *polymorph* and *ncompress* (stack underflow), the rest of the programs produced correct outputs in the validation step. In the case of the stack underflow bug in *ncompress*, a single prediction is isolated. However, after nullifying this instruction during validation, the program fails with a different stack trace. Therefore, the outcome of the validation stage for this bug prediction is labeled as unknown. As described in Section 4, we use the validation stage to rank the isolated anomalies. Since there is a single isolated anomaly, its rank remains as 1 and the faulty code is still successfully pinpointed. In *polymorph* memory is corrupted from two different locations and the two instructions need to be nullified together in order for the crash to disappear. In *gzip*, DIDUCE validates a different anomaly from AccMon and LoopCount. AccMon and LoopCount both detect the bug root-cause. When nullifying the root-cause, we prevent the buffer overflow, and the application succeeds. In comparison, DIDUCE detects a violation in a function call to *free*, which ultimately crashes the program. When nullifying the anomaly signaled by DIDUCE, we allow the buffer overflow to occur, but we still prevent the application from crashing.

Second, combining multiple bug predictors improves bug-detection coverage. For example, DIDUCE is not able to detect some bugs in *gzip* and *ncompress*, while AccMon and LoopCount catch those bugs. On the other hand, DIDUCE is the only one that catches the defect in *gcc*. Third, large applications such as *gcc* cause the bug predictors to report many anomalies, which highlights that the traditional approaches based solely on anomaly detection are less practical for large applications. As shown in Table 2, even though DIDUCE signaled the violation, without our approach DIDUCE will not be able to pinpoint the root cause since it is buried in too many (hundreds of) false positives. Next, we present a detailed case study on *gcc*, as it reveals some interesting aspects of our proposed approach.

6.1. Case Study: *The gcc 2.95.2 Compiler*

The *gcc 2.95.2* compiler has a defect, which causes the compiler to crash when compiling the program 'fail.c' with optimizations. The program 'fail.c' is shown in Figure 4.

The root-cause of the failure is a function call to *apply_distributive_law* in combine.c: lines 4013-4018, listed in Figure 5. The call to *apply_distributive_law* transforms expressions of the form (MULT (PLUS A B) C) to the form (PLUS (MULT A C1) (MULT B C2)), see Figure 6 (a) and (b). The problem is that C1 and C2 share a common grandchild (the macro XEXP(x, 1)) and thus they create a cycle in the abstract syntax tree, Figure 6 (c). Subsequent versions of *gcc* have fixed this defect by calling the *apply_distributive_law* function with a copy of C2 to prevent the common grandchild: *copy_rtx* (XEXP (x, 1)). Because of the cycle in the abstract syntax tree, the *gcc* compiler plunges into an infinite recursion loop in the function *if_then_else_cond* in combine.c: lines 6757-6788. The infinite recursion loop consumes so much stack space that eventually causes the operating system to terminate *gcc*. Using the call stack trace, we identified the function *if_then_else_cond* as the one in the infinite recursion loop.

This function constitutes the failure point of the program, and thus during automated debugging we examine the function call parameters for the token.

```
1  double mult(double z[], int n){
2    int i, j;
3
4    i = 0;
5    for(j =0; j<n; j++){
6      i = i + j + 1;
7      z[i] = z[i] * (z[0] + 1.0);
8    }
9    return z[n];
10 }
```

Figure 4. The fail.c program causes *gcc 2.95.2* to crash.

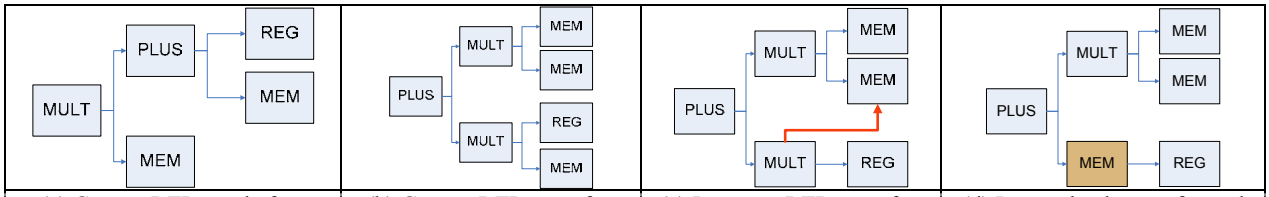
```
4009   case MULT:
4010   /* If we have (mult (plus A B) C), apply the
          distributive law and then the inverse
          distributive law to see if things simplify. */
4011    if (GET_CODE (XEXP (x, 0)) == PLUS)
4012    {
4013      x = apply_distributive_law
4014        (gen_binary (PLUS, mode,
4015            gen_binary (MULT, mode,
4016                XEXP (XEXP (x, 0), 0), XEXP (x, 1)),
4017            gen_binary (MULT, mode,
4018                XEXP (XEXP (x, 0), 1), XEXP (x, 1))));
       /*defect: causes a cycle in the abstract syntax tree */
4019
4020      if (GET_CODE (x) != MULT)
4021        return x;
4022    }
4023    break;
```

Figure 5. GCC defect: the call to apply_distributive_law creates a cycle in the RTL tree.

Zeller [37] showed that removing '+ 1.0' on line 6 from 'fail.c', makes the failure disappear. We used this passing input, as well as several other random C programs to train the bug predictors. After the training phase, we ran *gcc* on fail.c. DIDUCE produced 768 anomalies and 743 of them were 'new-code' anomalies. Since the failure point carried the token, we continued with the next step of our approach: automatic isolation of relevant bug predictions. After 571 delta-debugging runs, the number of anomalies was reduced to 84. Each of those 84 anomalies propagates the token to the failure point and constitutes a hypothesis for the root cause of the program failure. In the third step, we automatically validated each of these hypotheses. After the validation step, the 84 anomalies were classified as follows: application succeeds 2, application does not crash 9, unknown 28, and application fails 45. Nullifying the results of the 2 successfully validated instructions breaks the cycle in the abstract syntax tree and *gcc* does not enter into infinite recursion. Moreover, *gcc* produces a correct and working executable program. One of the validated anomalies corresponds to the root cause on line 17 in Figure 5. The other one is also involved in the construction of this portion of the abstract tree, which is the reason why it also breaks the cycle. Thus, we reduce the number of DIDUCE violations from 768 to only 2. To understand why *gcc* produces correct outputs in those two cases, consider Figure 6 again. The buggy version of *gcc* transforms the RTL tree as shown in Figure 6 (a) and (c). However, due to dynamically nullifying a certain instruction, the construction of the RTL tree remains incomplete, Figure 6 (d). *Gcc* iterates over the RTL tree multiple times and performs simplifications recursively, maintaining an undo buffer for each transformation. After a simplification, the resulting expression is evaluated to determine if it is still valid and if the simplification is profitable. If the simplified expression is found not to be valid or not profitable, then it is restored to its original state from the undo buffer. During our validation experiment, the incomplete transformation of the RTL tree is undone, and gcc produces correct code. In comparison, during an unmodified gcc execution, gcc plunges into an infinite loop while evaluating the RTL transformation and thus it is never able to undo the transformation. The same isolation and validation process was also automatically carried out for AccMon and LoopCount bug predictions. The number of AccMon anomalies was reduced from 1062 to 4 (17 does not crash and 4 produce correct outputs). The 666 LoopCount anomalies were reduced to 3. Thus, the number of relevant predictions was reduced from (768+1062+666, 2430 unique ones) to only (2+4+3, 9 unique ones). This example demonstrates that our approach is scalable to large software programs, and is able to pinpoint the defect among only 9 lines of code.

6.2. Impact of Hardware Implementation

Our proof-of-concept implementation using binary instrumentation incurs large performance overhead, typically two or three orders of magnitude. This is due to heavy instrumentation for each dynamic instruction (including library code). Combining multiple bug predictors, further contributes to this problem. To eliminate such overhead, we promote architectural support, which fits nicely for our proposed approach. Here, note that since our approach uses delta debugging to isolate relevant



(a) Correct RTL tree before transformation | (b) Correct RTL tree after transformation | (c) Incorrect RTL tree after transformation due to bug | (d) Incompletely transformed tree due to bug validation

Figure 6. GCC RTL tree transformations before and after function call to "apply_distributive_law".

Table 3. Bug detection results with adaptive partitioning of the bug predictor tables. Applications are compiled with "-static" option and library code is monitored for anomalies.

Application	Initial Bug Predictions			Isolated Bug Predictions			Validated (Application Succeeds)			Defect Rank
	D	A	L	D	A	L	D	A	L	
bc-1.06	48	79	40	6	3	4	1	1	1	**1**
gzip-1.2.4	66	62	30	2	1	1	1	1	1	**2**
ncompress-4.2.4 (*strcpy* defect)	7	6	6	0	1	1	0	1	1	**1**
ncompress-4.2.4 (stack underflow)	7	1	n/a	1	1	0	0	0	0	**1**
polymorph-0.4.0	24	10	20	4	1	0	0	0	0	**4**
man-1.5h1	31	115	36	3	2	0	1	1	0	**1**
gcc-2.95.2	210	380	424	17	38	32	1	4	1	**6**

anomalies, multiple debugging runs are required. This overhead is not our major concern since the purpose of automated debugging is to use computers to relieve software developers of this tedious job. Instead, we focus on the performance overhead of each debugging run as it may be critical in reproducing timing-related bugs.

Our proposed architectural support reuses the existing or previously proposed hardware structures in novel ways for debugging. Therefore, rather than presenting a detailed evaluation of hardware implementation issues such as area, latency or power, we analyze the impact of limited hardware resources on bug detection capability and show how our adaptive partition proposal in Section 2.2 solves the problem. In this experiment, we use 2k-entry prediction tables. If we do not apply our adaptive partition scheme, the debugging capability is impaired significantly due to frequent replacements, which may result in a high number of false positives or may even miss the actual root cause. For example, in *gcc*, the 2k-entry DIDUCE bug detector reports a total of 16,671 anomalies. Among those anomalies, many are detected as 'new code' violations incorrectly since the information of the executed dynamic instances are replaced. Such 'new code' violations further complicate the subsequent isolation or validation steps since all their dynamic instances need to be examined. To eliminate this adverse resource limitation impact, our proposed partition scheme tracks the number of replacements and adaptively partitions the code into a different number of regions, which are then monitored separately. This way, we can effectively reduce the resource requirement of the bug detectors. The bug detection results using our proposed adaptive partition scheme are reported in Table 3. Compared to Table 2, we can see that the number of initial bug predictions still varies. The reason is that with adaptive partitioning, the code is divided into only two or four regions for the BugBench applications, which under-performs a large 64K-entry table. On the other hand, in *gcc*, the code was partitioned into sixty-four regions, which has fewer replacements and false-positives than a 64K table. However, the differences in initial bug predictions are quickly smoothed away after the isolation and validation steps, where the false-positives are discarded and the actual defects are ranked.

6.3. Comparison to Other Approaches

In this section, we compare our proposed approach to a state-of-art debugging technique based on failure-inducing chops [10]. In this technique, the minimum failure-inducing inputs are isolated using delta-debugging [40]. Then, a dynamic forward slice originating from the minimum failure-inducing input is created. The forward slice is intersected with the dynamic backward slice originating from the program failure point, to obtain a chop. The instructions in the resulting chop are relevant to both the failure-inducing input as well as the failure point, and thus are likely to contain the program defect. We implemented the chop, by using only the dynamic data slices and ignoring control dependencies. For *bc-1.06* the defect was control dependent on the input, and so we manually expanded the slices to include the selected control dependences. A crash in Pin prevented us to obtain the chop for *man-1.5h1*. Because we only consider data dependencies, our resulting chop sizes are conservative, since the chops that we compute are a subset of the original chops. From the results presented in Table 4, we can see that our proposed approach pin-points the defect more accurately and presents the user a much smaller set of code to analyze. The reason is that our approach constructs dynamic slices originating from program anomalies rather than the program input. On the other hand, the failure-inducing chop approach is more general at the cost of requiring backward slicing and may find defects that escape our bug predictors. However, the large size of the failure-inducing chops, e.g., 1335 instructions in *gcc*, makes it very difficult for the user to analyze.

Table 4. Number of instructions in failure-inducing chops vs. the faulty code pinpointed by the proposed approach.

Application	Failure-Inducing Chops	Proposed Approach
bc-1.06	167	1
gzip-1.2.4	6	2
ncompress-4.2.4 (*strcpy* defect)	4	1
ncompress-4.2.4 (stack underflow)	11	1
polymorph-0.4.0	8	3
man-1.5h1	n/a	1
gcc-2.95.2	1335	9

7. Limitations and Future Directions

In this section, we highlight the limitations of our proposed automated debugging approach. First, the effectiveness of our scheme relies on the ability of bug predictors to signal relevant anomalies. If the defect is not signaled as an anomaly by the bug predictors, it will go undetected. As part of our future work, we are investigating the effects on program behavior caused by different types of software defects. One of them is invariance in redundant operations. It has been shown in previous work that redundant operations, such as impossible Boolean conditions, critical sections without shared state, variables written but never read, are likely indicators of software defects [34]. During our study with dynamic program execution, we observed a new locality that some instructions are very likely to produce redundant assignments, while others almost never result in redundant operations. Similar to other bug predictors, we can train a prediction table or a bloom filter to learn this locality. Then, any instruction performing an unexpected redundant operation will signal an anomaly. Our preliminary studies indicate that this approach can detect some bugs, including some logical ones from [2], which the other bug predictors fail to detect.

Second, in our current token tracking approach for bug isolation, only data dependencies are used to propagate the token. However, it is possible that an anomaly only leads to a branch condition and alters the control flow of a program. Since tokens are not propagated based on control dependencies, the token information may be lost in such cases. To address this problem, we can use confident branch mispredictions to filter this type of anomalies. In other words, a detected anomaly will be considered relevant only if it leads to a confident branch misprediction. Among the buggy code we examined, however, we have not found such a bug to evaluate this solution.

Third, the automatic verification approach can be further improved to serve as automatic program patches. As we could see from our gcc case study, about a third of the validation experiments resulted in an unknown state. Such unknown state is undesirable for systems that require failure-oblivious computing or self-healing. More intelligent approaches such as jumping to existing error handling code [28] may result in a safer program state.

Fourth, this paper shows that our proposed scheme is effective at debugging deterministic bugs. Further investigation on how to predict, isolate and validate concurrency bugs is part of our future work.

8. Related Work

There exists a rich body of research work to automate or facilitate software debugging. Due to space limitations, we briefly describe those works that are most closely related to ours and have not been previously described.

Anomaly Detection Dynamic program invariants were introduced in [8, 9] to facilitate program evolution and detect software defects. DIDUCE [11] and AccMon [44], as described in Section 3, exploit a compact representation of value-based or store-set invariants. Program anomalies have been shown useful to detect inconsistent use of locks [27] or atomicity violations [17, 18] in multithreaded programs. In [24], dynamic invariants have also been used in detecting and filtering soft errors.

Code coverage or spectra between passing and failing runs [12, 25] has been used for software debugging based on the observation that code executed only during the failing run(s) is more likely to contain software defects. The DIDUCE predictor that we use also has the capability to signal such 'new-code' anomalies, which combined with isolation and validation, were extremely helpful in pin-pointing the defect in *gcc*.

Dynamic Program Slicing Program slicing [31, 33] facilitates debugging, by presenting to the programmer all the statements which could possibly influence a variable of interest, and excluding the statements which are irrelevant. Dynamic program slicing [1, 14] includes all the statements which influence a variable of interest during a specific program run. Dynamic slicing typically results in a much smaller number of relevant statements than static slicing, but may still require the programmer to examine a significant portion of the program to locate the defect. To address this problem, a confidence mechanism is proposed in [41] to prune dynamic backward slices. The insight is that a statement that leads to the failure point may also produce correct values before the failure. The confidence of a statement is then computed from the profile of how likely it produces the incorrect values. Our approach is most closely related to failure-inducing chops [10], which we discuss in Section 6.3.

Delta Debugging Delta debugging is an automated process to isolate differences (deltas) between a passing and a failing run. The delta-debugging algorithm was first introduced by Zeller and applied to automatically isolate the failure inducing changes between an old and a new version of a program [38]. Subsequently, delta debugging is used to isolate and simplify failure-inducing input [40], to isolate failure inducing differences in program state [37], and to obtain cause-effect chains [4] that lead to the program failure. In our work, we apply the delta-debugging algorithm to isolate relevant bug predictions. Recent advances to speed up delta debugging [20] can also be used to improve our bug isolation process.

Nullifying Instructions Concurrently to our work, D. Jeffrey et al. [14] proposed to suppress/nullify memory writes to detect memory corruption bugs. In comparison, our approach is more general since we nullify instructions to validate various bugs and not only memory corruption. Also, nullifying is one step of our proposed approach.

Architectural Support Recently, a growing interest in architectural support for software debugging has been observed. iWatcher [43] exploits architecture support to implement flexible watch points to monitor program execution. Given the difficulty of reproducing failures, especially synchronization problems in multithreaded

applications, hardware assisted checkpoint-replay schemes [13, 13, 21, 22, 23, 29, 35, 36] have been proposed for deterministic replay of faulty runs. Although our work focuses on different aspects of software debugging, it benefits from these schemes as reproducing program failures is essential for any automated debugging process.

9. Conclusions

In this paper, we present a novel, automated approach to pinpoint the root causes of software failures. Our approach consists of three main components. First, we use a set of bug predictors to detect anomalies during program execution. Second, among the detected anomalies, we automatically isolate only the relevant ones. To achieve this, we construct the forward slices of anomalies to determine if they lead to the failure point. Each of the isolated anomalies then forms a hypothesis for the root cause of the failure. Third, we validate each hypothesis by nullifying the anomalous execution results. If the failure disappears, we can be confident that we have pinpointed the defect or the bug infection chain. We demonstrate that our approach is very accurate in pin-pointing the defects in all seven applications that we tested, and also outperforms existing state of the art debugging techniques. Further, we show that our approach is scalable to large software programs, such as the *gcc* compiler.

Acknowledgements

We thank Luis Ceze and the anonymous reviewers for helping us improve this paper. This research is supported by an NSF CAREER award CCF- 0747062.

References

[1] H. Agrawal and J. Horgan, "Dynamic program slicing", *PLDI*, 1990.
[2] A. Barr, "Find the Bug", *Addison-Wesley*, 2004.
[3] G. Chrysos and J. Emer, "Memory dependence prediction using store sets", *ISCA-25*, 1998.
[4] H. Cleve and A. Zeller, "Locating Causes of Program Failures", *ICSE*, 2005.
[5] J. Crandall and F. Chong, "Minos: Control data attack prevention orthogonal to memory model", *MICRO-37*, 2004.
[6] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security", ISCA-34, 2007.
[7] M. Dimitrov and H. Zhou, "Unified architectural support for soft-error protection or software-bug detection", *PACT-16*, 2007.
[8] M. Ernst, J. Cockrell, W. Griswold and D. Notkin, "Dynamically discovering likely program invariants to support program evolution", *IEEE TSE,* Vol.27, No. 2, Feb 2001.
[9] M. Ernst, A. Czeisler, W. Griswold and D. Notkin, "Quickly detecting relevant program invariants", *ICSE*, 2000.
[10] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating Faulty Code Using Failure-Inducing Chops", *ASE*, 2005.
[11] S. Hangal and M. Lam, "Tracking down software bugs using automatic anomaly detection", *ICSE*, 2002.
[12] M.J. Harrold, G. Rothermel, K. Sayre, R. Wu and L. Yi, " An empirical investigation of the relationship between spectra differences and regression faults", *Journal of Software Testing and Reliability*, Vol. 10, No. 3, 2000
[13] D. Hower and M. Hill, "Rerun: exploiting episodes for light weight memory race recording", *ISCA-35*, 2008
[14] D. Jeffrey, N. Gupta and R. Gupta, "Identifying the root causes of memory bugs using corrupted memory location suppression", *ICSM* 2008
[15] B. Korel and J. Laski, "Dynamic program slicing", *Information Processing Letters (IPL),* Vol. 29, No. 3, 1988
[16] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools", Work. on Eval. of SW Defect Detection Tools, June 2005.
[17] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: detecting atomicity violations via access interleaving invariants", *ASPLOS*, 2006
[18] B. Lucia, J. Devetti, K. Strauss, and L. Ceze, "Atom-Aid: Detecting and Surviving Atomicity Violations", *ISCA-35, 2008.*
[19] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation", *PLDI*, 2005.
[20] G. Misherghi and Z. Su, "Hierarchical delta debugging", *ICSE*, 2006.
[21] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: recording and deterministically replaying shared-memory multiprocessor execution efficiently", *ISCA-35*, 2008
[22] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging", *ISCA-32*, 2005.
[23] M.Prvulovic and J. Torrellas, "ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes", *ISCA-30*, 2003.
[24] P. Racunas, K. Constantinides, S. Manne, and S. Mukherjee, "Perturbation-based fault screening", *HPCA-13*, 2007
[25] M. Renieris and S. Reiss, " Fault localization with nearest neighbor queries", *ASE*, 2003
[26] H. Saal and I. Gat, "A hardware architecture for controlling information flow", *ISCA-5*, 1978
[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs", *ACM TCS*, 1997
[28] S. Sidiroglou, M. Locasto and A. Keromytis, "Hardware support for self-healing software services", *Workshop on Arch. Supp. for Security and Anti-Virus*, 2004.
[29] D. Sorin, M. Martin, M. Hill and D. Wood, "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery", *ISCA-29*, 2002.
[30] G. E. Suh, J. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking", *ASPLOS*, 2004.
[31] F. Tip, "A survey of program slicing techniques", *Journal of Programming Languages*, Vol.3, No. 3, 1995.
[32] N. Wang and S. Patel, "ReStore: Symptom Based Soft Error Detection in Microprocessors", *DSN*, 2005.
[33] M. Weiser, "Program Slicing", *IEEE TSE*, Vol. SE-10, No. 4, 1982.
[34] Y. Xie and D. Engler., "Using redundancies to find errors", *ACM SIGSOFT Symp. on Foundations of Software Engineering (FSE)*, 2002
[35] M. Xu, R. Bodik, and M. Hill, "Regulated Transitive Reduction (RTR) for Longer Memory Race Recording", *ASPLOS* 2006.
[36] M. Xu, R. Bodik, and M. Hill, "A flight data recorder for enabling full-system multiprocessor deterministic replay", *ISCA-30*, 2003.
[37] A. Zeller, "Isolating cause-effect chains from computer programs", *FSE-10*, 2002
[38] A. Zeller, "Yesterday my program worked. Today, it does not. Why?", *FSE-7*, 1999
[39] A. Zeller, "Why programs fail: a guide to systematic debugging", *Morgan Kaufmann*, 2005.
[40] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input", *IEEE TSE*, Vol. 28, No. 2, 2002. X.
[41] X. Zhang, N. Gupta and R. Gupta, "Pruning dynamic slices with confidence", *PLDI*, 2006.
[42] X. Zhang and R. Gupta, "Cost effective dynamic program slicing", *PLDI*, 2004.
[43] P. Zhou, F. Qin, W. Liu, Y. Zhou, J. Torrellas, "iWatcher: efficient architectural support for software debugging", *ISCA-31*, 2004.
[44] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S.Midkiff and J. Torrellas, "AccMon: Automatically detecting memory-related bugs via program counter-based invariants", *MICRO-37*, 2004.