Architectural Support For Improving System Hardware/software Reliability

2010

Martin Dimitrov
University of Central Florida

ARCHITECTURAL SUPPORT FOR IMPROVING SYSTEM HARDWARE/SOFTWARE RELIABILITY

by

MARTIN DIMITROV
B.S. Bethune-Cookman University, 2004
M.S. University of Central Florida, 2006

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2010

Major Professor:
Huiyang Zhou

ABSTRACT

It is a great challenge to build reliable computer systems with unreliable hardware and buggy software. On one hand, software bugs account for as much as 40% of system failures and incur high cost, an estimate of $59.5B a year, on the US economy. On the other hand, under the current trends of technology scaling, transient faults (also known as soft errors) in the underlying hardware are predicted to grow at least in proportion to the number of devices being integrated, which further exacerbates the problem of system reliability.

We propose several methods to improve system reliability both in terms of detecting and correcting soft-errors as well as facilitating software debugging. In our first approach, we detect instruction-level anomalies during program execution. The anomalies can be used to detect and repair soft-errors, or can be reported to the programmer to aid software debugging. In our second approach, we improve anomaly detection for software debugging by detecting different types of anomalies as well as by removing false-positives. While the anomalies reported by our first two methods are helpful in debugging single-threaded programs, they do not address concurrency bugs in multi-threaded programs. In our third approach, we propose a new debugging primitive which exposes the non-deterministic behavior of parallel programs and facilitates the debugging process. Our idea is to generate a time-ordered trace of events such as function calls/returns and memory accesses in different threads. In our experience, exposing the time-ordered event information to the programmer is highly beneficial for reasoning about the root causes of concurrency bugs.

To my family

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Huiyang Zhou for his mentoring, dedication, hard work, patience and motivation. Dr. Zhou has been a role model to me and has inspired me to never stop trying to grow and to improve. I feel extremely lucky to have Dr. Huiyang Zhou as my advisor.

This dissertation would not have been possible without the help and friendship of my fellow graduate students. Especially, I would like to thank Jingfei Kong, with whom we shared many fun and difficult moments during the last several years. I would also like to thank Slobodan Stipic for working with me and enduring with me through some of the hardest times in my graduate school.

I would like to thank my friends in the US and in Bulgaria for all the unforgettable memories that we have together, for their support in difficult times, and for being with me even when great distances have separated us. I would especially like to thank Vasil Niagolov for being such a great friend, even nine years after I have left Bulgaria to pursue my education.

I would like to thank all my great teachers, that I have taught me throughout the years, for inspiring me in one way or another to remain curious and to pursue education. I am very grateful to Dr. Dennis Clayton for his mentoring and motivation.

I would like to thank my girlfriend, Antoniya Petkova, for her love and understanding throughout my graduate studies. I would like to thank her for being by my side for the last six years.

Finally, I would like to thank my family for their love and support. My mother Snezhanka, my father Plamen and my sister Silva have always been my strongest support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

1.1. Hardware/Software reliability challenges

It is a great challenge to build reliable computer systems with unreliable hardware and buggy software. On one hand, software defects introduced by the programmer (commonly known as bugs) account for as much as 40% of system failures [33] and incur high cost, an estimate of $59.5B a year, on the US economy [41]. Some bugs are very difficult and time consuming to debug. For instance, memory corruption bugs may manifest only after a very long period of program execution or at unexpected locations. Concurrency related bugs may be difficult to reproduce and to reason about, due to their non-deterministic nature. Therefore, facilitating or automating the debugging process is an important step towards improving system reliability.

On the other hand, under the current trends of technology scaling, transient faults (also known as soft errors) in the underlying hardware are predicted to grow at least in proportion to the number of devices being integrated [63],[64] (i.e. with Moore's Law). Transient faults are caused by highly energetic particles passing through a semiconductor device. Such particles may include neurons from cosmic rays or alpha particles from packaging material. The charge carried by those particles may accumulate in the semiconductor device and eventually invert the state of the device, i.e. flip a bit in a memory cell or a logic gate from 0 to 1 or from 1 to 0. These faults are called transient, since they do not result in a permanent damage to the hardware and the correct state is restored on a subsequent write to the device. However, the corrupted bit may propagate to the program state and result in a program crash or even silently corrupted results.

1.2. Our solutions to improve system reliability

We propose several methods to improve system reliability both in terms of detecting and correcting soft-errors as well as detecting software bugs. Our first approach can be used flexibly for either soft-error protection or software bug detection. It is based on dynamically learning and enforcing instruction-level invariants. A hardware table is designed to keep track of run-time invariant information. During program execution, instructions compare their produced results against the learned invariants. Any violation of the predicted invariant suggests a potential abnormal behavior (i.e. an anomaly), which could be a result of a soft error or a software bug. When such abnormal behavior is detected promptly, simply nullifying the instructions in the processor pipeline (pipeline squashing) is able to fix most of the detected soft errors. The detected anomalies can also be reported to aid software debugging.

In our second approach, we improve anomaly detection for software debugging by detecting different kinds of anomalies and by removing false-positive anomaly reports. We use multiple anomaly detectors in order to detect more anomalies. Then, we remove false-positives by checking whether the dynamic forward slices of anomalies lead to the observed program failure. The dynamic forward slices are efficiently computed in hardware, by utilizing existing taint architectures. We also validate the remaining anomalies by dynamically nullifying their effects and observing if the program still fails. In the failure disappears after nullifying the anomalous instruction, we can be fairly confident that we have pin-pointed root cause of the bug.

While the anomalies reported by our first two methods are helpful in debugging single-threaded programs, they do not address concurrency bugs in multi-threaded programs. In our third approach, we propose a new debugging primitive which facilitates the debugging process

by exposing the non-deterministic behavior of concurrent programs to the software developer. Our idea is to generate a time-ordered trace of events such as function calls/returns and memory accesses in different threads. The architectural support for this primitive is lightweight, including a local counter in each processor core, a way to synchronize local counters to a global time-stamp, and event trace buffers. The proposed primitive can be used to record the last N events, or be directed through a flexible software interface. Our experience shows that exposing such time-ordered event information to the programmer is highly beneficial for reasoning about the root causes of concurrency bugs.

1.3. Contributions

The contributions of this dissertation are summarized as follows:

1. We propose a flexible unified architectural support for either soft-error protection or software bug detection. We consider our approach an information redundancy scheme (vs. time or space redundancy), which uses localities to encode the proper instruction execution and protect processor logic. Our scheme does not require any redundant execution, and thus it eliminates much of the power and performance overhead associated with space or time redundancy approaches. Our design opportunistically protects multiple processor structures including: decode logic, rename tables, the register file, issue queues, and functional units, significantly improving Mean Time to Failure (MTTF). For software bug detection, our architecture approximates previously proposed software-based bug detection approaches, while incurring less performance overhead.

2. We propose an automated approach, which improves on anomaly detection for software debugging and is used to pin-point 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 by constructing the dynamic forward slices of anomalies to determine if they lead to the failure point. Third, we validate each isolated anomaly by nullifying the anomalous execution results. If the failure disappears, we can be confident that we have pinpointed the defect or that we have broken the bug infection chain. We demonstrate that our approach is very accurate in pin-pointing the defects in our test applications, and also outperforms existing state of the art debugging techniques. Our approach can also be implemented with efficient architectural support.
3. We propose a debugging primitive to construct time-ordered event traces, which can be helpful in the debugging concurrency bugs. We design a light weight architectural support and a software interface to support the proposed primitive. We evaluate the proposed primitive with a variety of bugs from large production software (MySQL and Mozilla), including: deadlock, atomicity violations, order violations and logical concurrency bugs.

CHAPTER 2. UNIFIED ARCHITECTURAL SUPPORT FOR SOFT-ERROR PROTECTION OR SOFTWARE BUG DETECTION

While software bugs and soft-errors have unrelated origins, they manifest in similar ways. Both soft-errors and software bugs can cause a program to behave unexpectedly, to crash, or even to silently corrupt the output data. Even though both types of errors manifest in similar ways, previous work has treated the problems separately. In this chapter, we realize that by exploiting program localities, we can detect abnormal behavior in order to either protect processors from soft errors or to hunt down software bugs.

Historically, program localities have been studied extensively and widely used in high performance processor design. In this work we observe that program localities also enable exceptional behavior (i.e. anomaly) detection: if an instruction satisfies a certain pattern or a locality, any diversion from this pattern could indicate an anomaly. Here we focus on a value locality, named limited variance in data values (LVDV), to detect either soft-errors or software bugs. LVDV is based on the observation that the execution results of many instructions vary only within a certain, predictable range. In other words, if we compute the data variance of a static instruction by XORing its last two dynamic execution results, the variance is usually small, indicating that only a limited fraction of the result bits vary among different execution instances. The range of variance can be encoded as a signature of instruction execution. If the instruction produces a result, for which the variance exceeds the previously learned variance, we can speculate that an exceptional event has occurred. The cause of the anomaly can be a soft error induced by natural radiation sources or it can be a latent software bug introduced by programmers. We propose a simple, unified architectural support, which can be used flexibly to

either opportunistically protect the processor pipeline from soft errors, or to help developers track down the root causes of software bugs.

The proposed architectural support contains a hardware table, which tracks the variance of instructions' execution results. During program execution, instructions update the table with their computed results, while detecting whether the computed results violate the predicted variance. We allow different sets of instructions to update the table depending on whether soft-error or software bug detection is desired. In the first case, if the predicted variance is violated, we speculate that a soft error has occurred and squash the processor pipeline (i.e. nullify all instructions in the pipeline). The offending instruction, as well as the other squashed instructions, is re-executed in an attempt to correct the soft error transparently. In the second case, the persistent anomalies are logged to facilitate software debugging.

Compared to traditional soft-error protection approaches, which utilize space or time redundancy [3][19][37][52], we consider our approach an information redundancy scheme, which encodes the proper instruction execution to protect processor logic. Since it does not require any redundant execution, it eliminates much of the power and performance overhead associated with space or time redundancy approaches. Our design opportunistically protects multiple processor structures including: decode logic, rename tables, the register file, issue queues, and functional units.

For software bug detection, our architecture approximates the software-based bug detection approach DIDUCE [20]. However it also provides unique advantages compared to software-based bug-tracking approaches:

• Performance efficiency: Our approach is implemented in hardware and incurs minor performance degradation due to bug monitoring.

• Binary compatibility: Since it is a pure hardware scheme, our approach is language independent and works directly with the binary code without the need for recompilation.

• Runtime monitoring: Since our approach has very limited impact on performance, it is possible to use it after the software construction phase, or after the product has been released. In this scenario, invariant violation reports can be incorporated into tools such as Windows Error Reporting (WER) [79], to provide developers with additional information about program behavior or possible causes of an application failure.

2.1. Limited variance in data values

In this work we use the term Limited Variance in Data Values (LVDV) to describe the locality of instruction-level invariants. Variance between two values is simply defined as the result of XORing the two values and observing the most significant bit position containing a '1'. For instance a variance of 0x0000007F, means that variance between the values is constrained to the least significant 7 bits (since they contain '1', the rest of the 25 bits are all '0'). LVDV extends the traditional/classical value localities [26][55] and can be exploited for higher coverage and lower false-positive rates in terms of locality violations.

LVDV is based on the observation that for many instructions, even if they don't show predictable value patterns, the variance among their execution results is usually limited. For example, for an instruction with outputs: 1, 60, 122, 40, 402, 7, etc, variance it constrained to the least significant 9 bits although there seems to be no apparent value pattern. An output of 1000000014 (with variance in bit 30, compared to the previous results) hints at a high possibility of exceptional behavior. LVDV also captures the region locality, which refers to the fact that memory operations tend to access data in a fixed (or bounded) region. For example, a load

accesses a certain data structure in the heap space and it generates the following address sequence that has no stride locality: 0x11112654, 0x11117838,…, 0x11111200, 0x11119088, …. Then, an out-of-place address such as 0x01117854 (an address accessing the text segment) or 0x71117800 (a stack address) or 0x1191c014 (a seemingly out-of-range heap address) would indicate a likely error.

For instructions with traditional value localities, LVDV provides a more effective way of encoding their characteristics for violation detection. For example, for an instruction with a repeating stride pattern, 1, 2, 3, …100, 1, 2, 3, …,100, etc, the variance of the results is constrained to the lower 7 bits and any result showing a larger variance would signal a potential violation. Compared to the traditional stride value locality, although any error in the lower 7 bits cannot be detected by LVDV, the majority of data computation, which produces the upper 25 bits of the results, is protected (assuming a 32-bit machine). More importantly, LVDV eliminates all the false positives that would have been signaled using the stride value locality as the stride fails to characterize transition values (i.e. as the value changes from 100 to 1) correctly. Since soft errors / software bugs in production code happen rather infrequently, LVDV presents a more desirable tradeoff between protection coverage and performance overhead.

2.2. Related Work

2.2.1. Locality-based soft-error detection

Implicit redundancy through reuse (IRTR) [18] utilizes instruction reuse [59] for soft-error protection. IRTR stores both operation inputs and outputs in a reuse buffer (RB). When an instruction hits in the RB, its inputs are compared to the inputs stored from the previous execution of the same instruction. If the inputs match, then the result stored in the RB and the

currently computed result can be compared for error detection. With IRTR, the error detection is un-speculative and there are no false alarms if ignoring any possible soft errors in the RB. However, corruption of the input values, either in the RB or in the currently executing instruction will cause the input comparison to fail, resulting in a loss of coverage. Therefore, IRTR is not suitable for protecting input-related logic, such as the rename table or source operand decode logic. Our scheme protects more logic units since only the instruction PC is needed to check the expected variance. The storage overhead is also reduced compared to IRTR, since we do not need to keep input values.

Exploiting value locality for soft error detection bears similarity to symptom-based soft error detection, in which mispredictions of high confidence branches are used as symptoms of soft errors [63]. The advantage of exploiting value locality is that an error can be detected more promptly and simple pipeline squashing is likely to fix the error as shown from our experimental results.

Concurrently to our study, a similar idea was independently proposed by Racunas el at. [49]. Racunas et al. uses a more generic approach and evaluates the tradeoffs of utilizing different events for soft-error detection. An architectural implementation, similar to ours, is also proposed and evaluated. In this work, we also advocate the use of program localities to detect errors, but we focus on one particular locality, namely LVDV. We provide an in-depth analysis of the protection coverage provided to different hardware structures, including Issue Queues and Functional Units, and compare our approach to three other approaches.

2.2.2. Locality-based software bug detection

Program localities, invariants in particular, have previously been exploited by software-based approaches such as DAIKON [14][15] and DIDUCE [20] to discover software bugs. It has been shown that invariant violations (anomalies) are especially helpful to pinpoint latent code errors [20]. In these approaches, the program's source code or object code is instrumented and the results of selected static instructions or expressions are monitored in order to learn the invariants. Learning the invariants is accomplished by initially hypothesizing the strictest invariants, and then gradually relaxing the hypothesis as invariants are being violated. To minimize the overhead of tracking the invariant information, DIDUCE uses a single bit mask for each tracked expression. The bit mask indicates which bits of the expression have changed, compared to the previous executions of the same expression. The bit mask is computed by an XOR operation between the results of the current and the previous execution of the expression.

Our proposed approach can be viewed as a hardware implementation of DIDUCE. Our approach, requires no program instrumentation/recompilation and therefore it is binary compatible. It also eliminates the substantial performance overhead associated with the software-based approaches. Thus, it is capable of providing transparent and run-time bug monitoring.

Oplinger et al. [44] proposed to speed up the execution of monitoring functions (invariance checking or any other monitoring function) by executing the monitoring code in parallel to the main program using thread-level speculation (TLS). Compared to [44] our approach is more lightweight as it does not require binary instrumentation or significant hardware changes required by TLS.

Another approach taking advantage of architectural support to detect software bugs is AccMon [77]. AccMon exploits the store set locality of load instructions, i.e., a memory location is usually updated only by certain store instructions, to detect abnormal memory operations. Since AccMon and our proposed approach exploit different program localities, they are complementary to each other although some bugs can be detected by both approaches.

2.3. Proposed architectural support

We propose a hardware structure, named the LVDV table, to keep track of instruction-level invariants. As shown in Figure 1, the LVDV table is a cache structure. Each data entry in the table contains a variance field, a last-value field, and a K-bit saturating confidence counter. To reduce the storage overhead, we propose the following encoding mechanism for variances. A 32-bit variance is first divided into N equal chunks. If all the bits in a chunk are zeros, a bit '0' is used to encode the entire chunk. If any of the bits in a chunk is '1', a bit '1' is used to encode the chunk. In this way, any variance can be encoded in N instead of 32 bits. The decode process is straightforward. For example, when N equals 4, the encoded value '001x' is simply decoded to a 32-bit variance 0x0000FFFF, meaning that the variance should be constrained within the lower 16 bits or lower two chunks.

Instructions access the LVDV table with their program counter (PC). The variance between the instruction's last two results is obtained by XORing the current execution result and the last value from the LVDV table. The variance is then compared with the encoded variance. If the current variance is larger than the encoded one and the confidence counter is above a set threshold, an anomaly is detected. If the current variance is larger than the encoded one and the confidence is low, that means that the LVDV table is still learning the proper range of the

variance. The current larger variance then replaces the stored one and the confidence counter is reset. If the current variance is smaller than or equal to the encoded one, the confidence counter is incremented by one and there is no update to the stored variance. As a last step, the last value is replaced with the current execution result.

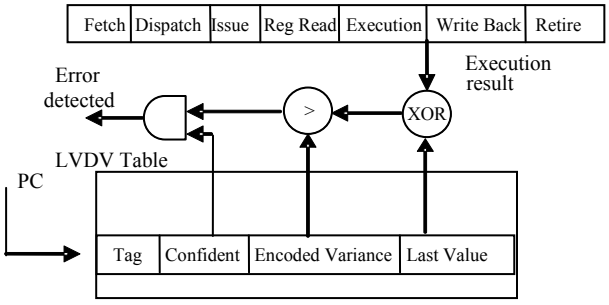

Figure 1. The architecture to exploit LVDV for soft error detection or software bug detection.

2.4. Soft-error protection

In this section we address how we use the LVDV locality to detect/recover from soft-errors. We also present our experimental results to show the effectiveness of the proposed approach compared to other soft error detection schemes.

2.4.1. Soft-error recovery mechanism

In this section we address how we use the LVDV locality to detect/recover from soft-errors. We also present our experimental results to show the effectiveness of the proposed approach compared to other soft error detection schemes.

The LVDV table maintains the variances of value-producing instructions, except memory operations, for which the variances of the addresses are encoded. Although load values are not protected directly in this way, immediately dependent operations offer indirect protection if they

exhibit limited variances. When a likely soft error is detected by the LVDV table, the processor can fall back to a previous checkpoint as proposed in ReStore [63]. Alternatively, it may squash the pipeline and resume execution from the instruction that resides at the head of the re-order buffer (ROB). In this work, we adopt pipeline squashing for its simplicity and our experimental results show that pipeline squashing is capable of fixing many errors that occur in the issue queue or functional units. The reason is that an error is promptly detected if the faulting instruction or one of its immediately dependent instructions has limited variance. In such cases, pipeline squashing is sufficient to prevent the error from being committed to the architectural state and the re-execution of the faulting instruction ensures correctness. In case the detected anomaly is a false positive, pipeline squashing incurs performance overhead but does not affect correct program execution.

The LVDV table captures instruction-level execution behavior. Therefore, a single LVDV table is capable of detecting any soft error which occurs in the pipeline as long as the altered execution results lead to a higher-than-expected variance. Besides the computational logic in the execution stage, control logic such as the decoder, renaming table, issue queue, and operand selection logic are protected. In our experiments Section 2.6, the protection of the issue queue and functional units are examined in detail.

2.4.2. Reliability and Complexity Impact of the LVDV Table

In this section we elaborate on several issues related to the implementation of the LVDV table. We address the effects of soft errors occurring in the LVDV table itself, the impact on cycle time, and the ways to improve the variance encoding techniques.

Like any logic units in the processor, the LVDV table is susceptible to soft errors but there is no need for any protection for the LVDV table. The reason is that soft errors, which corrupt the LVDV table, have two possible outcomes: they either induce the LVDV table to signal a false-positive anomaly, or result in a loss of error coverage. A soft error occurring in a confidence counter, for example, may set the counter to be confident prematurely. In this case, the LVDV could be prompted to signal an anomaly, while in fact it should be still learning the proper variance of this instruction. On the other hand, a soft error lowering the confidence counter will simply delay the learning process for that instruction slightly. A soft error in the "Variance" or "Last Value" field in an LVDV entry can also cause a false-positive anomaly alert- for example by lowering the variance to a lower chunk. On the other hand, the "Variance" or "Last Value" could be corrupted in such a way as to limit the error coverage for a particular entry. This could happen if the soft error moves the variance to a higher chuck. Similar false positive or loss of coverage interactions are possible if the error occurs in the "Tag" field as well. To prevent accumulation of errors in the LVDV table, which result in loss of coverage, the table is simply flushed periodically.

The design presented in Figure 1 can be tuned with the following optimization. Rather than computing the variance between execution results directly, we can first compute the DELTA (Δ) between execution results and then compute the variance between two DELTAs. The advantage of this optimization is that for some value sequences, the range of the variance can be significantly reduced when the variance is computed on their DELTA sequences. The overhead is that it needs extra hardware to compute subtraction and requires an extra field in the LVDV table to store DELTA along with the last value. However, our experiments with this

DELTA variance optimization do not show sufficient improvement in error detection to justify the overhead.

The LVDV table only needs the instruction PC in order to start the access. The instruction PC is available as early as the fetch stage, while the only requirement on the LVDV table is that the access is complete by the end of execution stage. Therefore, the LVDV table is not on the critical path of the processor and should not impact the cycle time.

2.4.3. Fault-injection methodology

We evaluate the effectiveness of our mechanism using fault injection. Errors are injected into the issue queue (IQ) and the functional units (FUs) of our microprocessor model. Errors do not occur unless they are purposely injected by us. The protection level of either structure is evaluated separately by performing 10 runs and injecting at least 10000 errors per run into the structure under study. According to the analysis in [63], 10000 per run is a large enough number of injections to make our results statistically significant. Similar to [63] we pre-compute a list of random cycles at which to cause a single-event upset. Upon reaching a designated cycle, a random bit is flipped into the target structure. After injecting a fault, we let the error propagate using execution-driven timing simulation. We simulate 10000 cycles after the fault is injected based on the condition that the control flow is not altered and there are no exceptions such as memory access violations. At the end of the 10000-cycle trial period, the architectural state including the program counter, the architected register file, and memory are compared against a fault-free model. If a mismatch is detected, then we assume that the error will not be masked and is critical. On the other hand, if no mismatch is detected, then the error must have been either masked during normal program execution (i.e., a dead or unused bit is flipped) or fixed by some

fault protection mechanism. During the trial period, if the control flow deviates from the fault-free model (i.e., a retiring branch jumps to the wrong target) or a memory access violation is detected, the error is determined to be unmasked and critical. After exiting the trial period, the timing simulator restores the architectural state from the fault-free model and resumes normal simulation until it reaches the next designated fault-injection cycle.

When injecting errors into the issue queue (IQ), we target all the instructions' source and destination operands and opcode. Errors are not injected in any of the additional state bits kept in the IQ, such as bits which indicate if an operand is ready. A soft-error which marks an operand as not-ready may cause a deadlock, which is easily detected by a watchdog timer and thus we ignore such errors. Due to lack of circuit implementation details in our timing simulator, we cannot properly model error propagation within combinational logic units. Therefore, when injecting faults into the functional units, we flip a bit in the final computed result. This is sufficient for our purposes, because we are only interested in determining how many of the errors which propagate from the FUs can be removed by the proposed mechanisms.

In order to evaluate the effectiveness of a fault protection scheme, we first perform fault injections without any error protection (i.e., the base case) and record the number of critical faults (i.e., faults that are not masked). Then, with a fault-protection mechanism enabled, we repeat the fault injection campaign and record the number critical faults again. The difference in the number of critical faults shows the effectiveness of the fault-protection scheme.

Compared to our preliminary study [12], we perform a larger number of injection runs in this work. This is because the number of reported critical faults may vary by up to 10-13% between runs as a result of random fault injection. By averaging the results of multiple runs, we

eliminate much of this random effect. In addition, the injections in this work are more accurate since every bit in the structures under study is accounted for.

2.5. Processor Model

Our simulator models an MIPS R10000 style superscalar processor and its configuration is shown in Table 1. All the experiments are performed using SPEC CPU 2000 benchmarks with the reference inputs. Representative simulation points are determined using the SimPoint [56] with the program phase size as 600M instructions given the requirements set by our fault injection methodology.

Table 1. The configuration of processor model.

Pipeline	3-cycle fetch stage, 3-cycle dispatch stage, 1-cycle issue stage, 1-cycle register access stage, 1-cycle retire stage. Minimum branch misprediction penalty = 9 cycles
Instruction Cache	Size=32 kB; Assoc.=2-way; Replacement = LRU; Line size=16 instructions; Miss penalty=*10* cycles.
Data Cache	Size=32 kB; Assoc.=2-way; Replacement=LRU; Line size = 64 bytes; Miss penalty=*10* cycles.
Unified L2 Cache (shared)	Size=1024kB; Assoc.=8-way; Replacement = LRU; Line size=128 bytes; Miss penalty=*220* cycles. Stream buffer hardware prefetcher.
Branch Predictor	64k-entry G-share; 32k-entry BTB
Superscalar Core	Reorder buffer: 128 entries; Dispatch/issue/retire bandwidth: 4-way superscalar; 4 fully-symmetric function units; Data cache ports: 4. Issue queue: 64 entries. LSQ: 64 entries. Rename map table checkpoints: 32
Execution Latencies	Address generation: *1* cycle; Memory access: *2* cycles (hit in data cache); Integer ALU ops = 1 cycle; Complex ops = MIPS R10000 latencies

The LVDV table has a default size of 2048 entries and is configured as 4-way set-associative. Each entry in its data store takes 43 bits, including a 3-bit confidence counter, an 8-bit variance value (i.e., we use 8 chunks to encode the 32-bit variance), and a 32-bit field for the last value. Therefore, the overall size of the LVDV table is 88k bits (or 11.008 k Bytes).

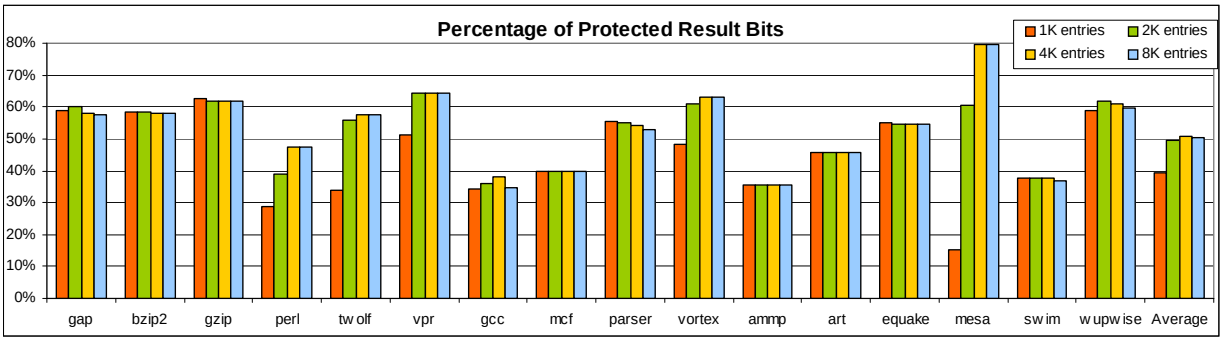

Figure 2. The fraction of protected bits using LVDV locality.

2.6. Experimental Results

2.6.1. Strength of the LVDV locality

We first examine the strength of the LVDV locality by checking the fraction of bits in execution results that are protected using our LVDV scheme. For a result with variance constrained within the lower k bits, the remaining (32-k) bits of the result are protected. We varied the LVDV table size from 1K entries to 8K entries and used 8 chunks to encode the 32-bit variance (8 chunks means that we use 8 bits to encode a 32 bit value). We also experimented with different number of chunks and determined that 8 chunks provide a good balance between protection coverage and low false-positive rate. The ratio of all the protected bits over the overall result bits is reported for each benchmark, as shown in Figure 2. From the figure, we can see that the proposed LVDV protects a significant portion of execution results, up to 80% in *mesa* and 50% on average for an 8K entries LVDV table. Second, we observe that for some benchmarks, such as *perl*, *twolf*, *vpr* and *mesa*, LVDV provides much better protection once the working set of the application fits into the LVDV table. In *mesa*, protection varies from 15% to 80% for a 1K and 8K entries table respectively. However, it is interesting to observe, that in some cases such

as *parser* and *gap*, increasing the LVDV table size results in slightly decreased protection. This happens because some entries are rarely evicted from a large table and once the variance of a static instruction is learned, it is never reset. We observed that in some cases it is beneficial to periodically reset the learned variance, which may becomes overly conservative due to wide variations in execution results. A small LVDV table will frequently replace entries due to conflicts and thus refresh their variance information and enable more execution result bits to be protected. Among all the examined sizes, we observe that for most benchmarks a 2K-entry LVDV table provides comparable protection to an 8K LVDV table. Therefore, we use a 2K entries table with 8 chunks as default configuration for our soft-error protection experiments.

While effective at capturing localities for integer or address computation, it is harder for LVDV to capture localities for floating-point computations. Such computations are usually performed with 64-bit doubles, which consist of 1 sign bit, 11 exponent bits and 52 mantissa bits. In our LVDV table, we keep track of only 32-bit execution results, and therefore we choose to protect only the top 32 bits of large floating point values. This way, we keep track of the variance of the sign bit, the exponent and 20 of the mantissa bits. In our experiments, we observed that the variance of the mantissa is quite unpredictable and in most cases no protection is provided. On the other hand, LVDV is able to protect 3 out of 11 exponent bits on average for the floating point benchmarks, and up to 10 out of 11 for *equake* and *mesa*.

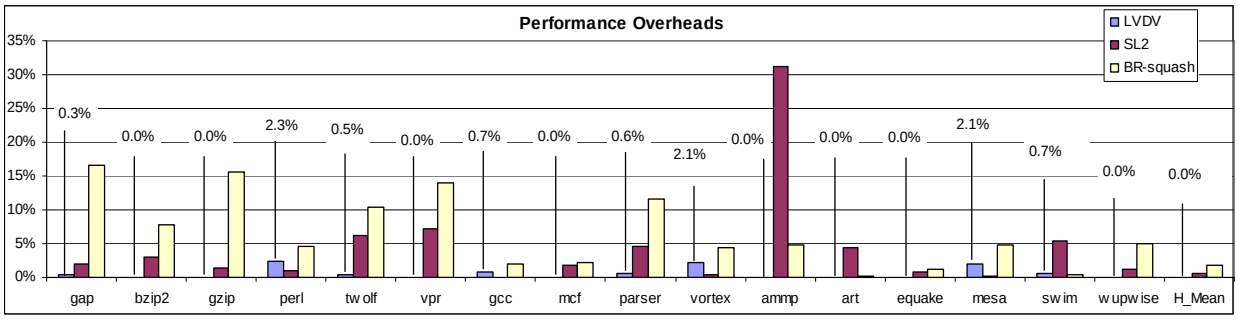

Figure 3. Performance overheads of different error protection schemes

2.6.2. Analysis of performance overhead

We first evaluate the performance overheads introduced by different protection mechanisms in fault-free environment. We compare our approach to Squash on L2-miss (SL2) [64] and Branch-miss Squash (BR-squash) [63]. The Instruction Redundancy through Reuse (IRTR) [18] approach, detailed in Section 2.2.1, is un-speculative and does not incur performance overheads. The idea of SL2 is to keep critical data away from vulnerable structures. SL2 provides partial protection to the IQ by squashing instructions when a long latency L2-cache miss is being repaired. The rationale is that instructions in the IQ are unnecessarily exposed to soft errors while the pipeline is essentially idle. We implemented SL2 by performing a complete pipeline squash whenever the ROB is full and the instruction at the head of the ROB is detected to be an L2 cache miss. The pipeline resumes fetching instructions as soon as the L2 cache miss has been repaired. In [12], SL2 is implemented by squashing the pipeline as soon as the instruction at the head of the ROB is known to be an L2 cache miss (without waiting for the ROB to become full). Such more aggressive squashing resulted in higher protection coverage for some benchmarks, but also led to larger performance penalties due to more frequent squashing. BR-squash is a modified version of the symptom based protection mechanism proposed in [63]. In the original symptom mechanism, when a confident branch is mispredicted, the processor is

rolled back to a previous checkpoint. In this work, we do not implement the checkpointing mechanism and simply squash the pipeline when a misprediction of a confident branch is resolved. The reason is to show how promptly the impact of a soft error can manifest in program execution. The branch prediction confidence is modeled by a 4k-entry table and each entry is a 3-bit saturating counter.

The performance results are show in Figure 3. Here, the average performance is computed as the harmonic mean of the IPCs and then normalized to the baseline processor (labeled as H_Mean). In SL2, instruction execution can be significantly delayed since squashing on an L2 cache miss may nullify many instructions, which are independent of the cache miss. For the benchmark *ammp*, many completed long-latency floating-point operations are squashed because of an independent cache miss, resulting in 31% performance degradation. On average, 0.5% slowdown is incurred by the SL2 approach. BR-squash also reports relatively high performance overheads for some benchmarks, up to 16.6% for *gap* and an average of 1.8%. BR-squash incurs higher overheads for the integer benchmarks due to their relatively high branch misprediction rates. The floating-point benchmarks have low branch misprediction rates and so the overhead is much lower as seen in Figure 3. The proposed LVDV scheme incurs very limited performance overhead, up to 2.3% in the benchmark *perl* and an average of 0.02%.

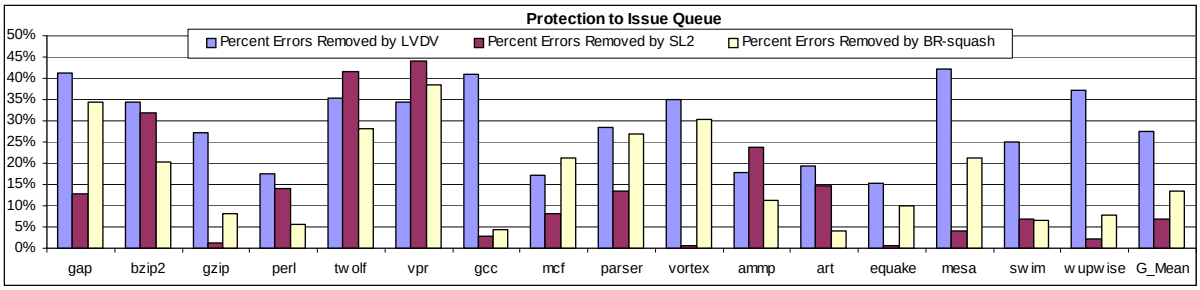

Figure 4. Protection to IQ by LVDV, SL2 and BR-squash.

2.6.3. Soft-error protection to issue queues

In Figure 4, we compare the protection provided to the Issue Queues by our approach to SL2 and BR-squash. LVDV performs the best by removing 28% of critical errors on average, compared to 7% and 14% for SL2 and BR-squash respectively. Removing 28% of critical errors translates to 39% improvement of MTTF (Mean Time to Failure), which is calculated as 1 / (1-% errors removed).

Notice that the LVDV locality is very general since it is able to provide reasonable protection across all the benchmarks. On the other hand, both SL2 and BR-squash are highly application specific, providing significant protection to some benchmarks (*twolf*, *vpr*) and almost no protection to others (*gcc*). In general, BR-squash is effective on benchmarks with a relatively high number of branch mispredictions, such as *gap*, *parser*, *twolf*, *vpr*, and *vortex*. For benchmarks with low branch misprediction rates, e.g., *gzip* and *gcc*, although many injected errors result in control flow errors, BR-squashing cannot fix them since it is too late to prevent the error from propagating to the architectural state when the misprediction is detected. Therefore, a checkpoint mechanism is necessary for BR-squashing to restore the architectural state. In comparison, LVDV detects errors more promptly and a simple pipeline squash can fix them in time. Similarly, SL2 is very effective in protecting those benchmarks, whose progress is frequently blocked by an L2 cache miss such as *twolf* and *vpr*, and offers almost no protection to other benchmarks such as *gzip*, *vortex*, *equake* and *wupwise*.

In our experiments, IRTR did not protect the IQ well. The reason is that our simulator models a MIPS R10000 style pipeline and its IQ does not contain the operand values. As errors are only injected to the opcode and operands, IRTR only protects the opcode. In a

microarchitecture that models the issue logic using reservation stations, IRTR will be more effective.

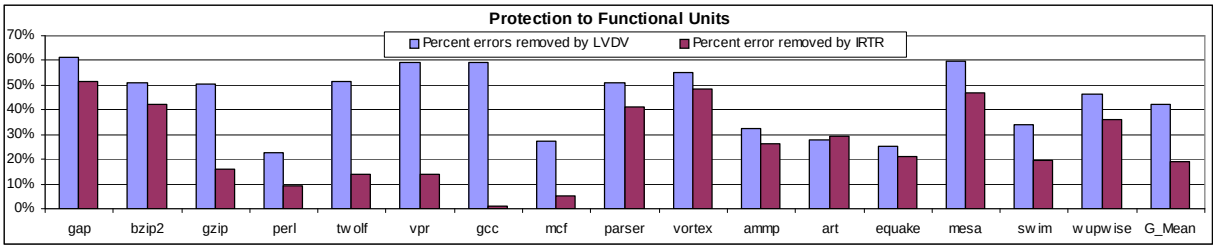

Figure 5. Protection to FUs by LVDV and IRTR.

2.6.4. Soft-error protection to functional units

In this experiment, we evaluate the effectiveness of LVDV on FUs as compared to IRTR and the protection coverage achieved by both schemes is reported in Figure 5. We implement IRTR as a 2048 entry, 4-way table. Each entry contains two inputs and one output, for a total of 192k bits. We do not include SL2 and BR-squash in this experiment as these mechanisms did not protect well from the faults injected into the FUs. From Figure 5 we see that the proposed LVDV removes many more critical errors than IRTR. It achieves a reduction of critical errors of up to 61% for *gap* and 42% on average. Considering the Mean Time to Failure (MTTF) of the FUs, our opportunistic error protection provides up to 156% improvement of MTTF for *gap*, and 72% improvement of MTTF on average. LVDV performs better than IRTR because it is able to extract useful locality information from every benchmark and protect a fraction of the result bits. On the other hand, IRTR protects all-or-none of the results bits and thus performs poorly for benchmarks with low instruction reuse locality.

2.7. Software bug detection

In this section, we elaborate on our proposed architectural support for software bug detection, including the implementation details and the experimental results with several applications.

2.7.1. Software bug detection mechanism

As addressed in Section 2.2.2, the proposed architectural support can be viewed as a hardware implementation of the statistics-rule-based software approach DIDUCE. In general, statistic-rule-based approaches [14][15][20][77] rely on extracting invariance information (or statistical rules) automatically from multiple successful program runs, or from the continuous execution of a single long run. Once the invariants have been obtained, they can be used to detect anomalies in subsequent runs. The invariants can also be used to detect anomalies within the same long program run once the rules are established. Statistic-rule-based approaches are promising because they can detect bugs that do not violate any programming rules [77]. For instance, a programming rule in C++ could be that "an array pointer should not move out of bounds". However, a bug which causes the array elements to be accessed in the wrong order (without going out of bounds) cannot be detected by a programming rule based approach.

Similar to other statistic-rule-based approaches, the usage of our proposed mechanism contains two phases: the training phase and the bug-detection phase. In the training phase, our LVDV table learns the invariant information from successful program runs or during a long execution run. To preserve invariance information across multiple program runs, we require the LVDV table to be written to a file at the end of each program run, and reloaded at the beginning

of a new run. During the bug-detection phase, the LVDV table is used to detect violations of the inferred invariance rules and any invariant violations (anomalies) will be output to a log file. The log for each anomaly includes the PC (program counter) of the faulting instruction, the previous and currently produced values, the predicted variance, and confidence. Also, any misses in the LVDV table can be reported as "new code", or instructions not executed during the training phase.

Due to the limited capacity of the LVDV table, it is possible for entries to be evicted and replaced from the table, which can result in two potential adverse effects: an increased number of false-positive alerts and a reduction in detection coverage. The first effect can be explained as follows. When new code is encountered, false positives are common since the proper range of variance has not been established. The replacement of entries from the LVDV table can create a similar effect, because the variance information of the replaced instruction has been discarded. In this case, it is possible to receive multiple violations with the same variance for the same static instruction. Fortunately, such replicate anomalies can be easily detected and removed by a simple post processing of the bug report (eliminate duplicate anomalies from the same instruction). The second concern originating from the limited LVDV table size is the loss of detection coverage. When the variance information of a static instruction is replaced from the LVDV table, it is possible that this information will not be available again in the table at the time of bug manifestation. To address the issue of limited table sizes, we allow only store instructions to access the LVDV table, and we track the variance of their addresses. The reason why this approach is effective is that most bugs manifest through memory operations [27]. Moreover, if the memory operation is at the end of a dependence chain, violations in previous dependent instructions are likely to propagate to the tracked memory operation. In our experiments, every

store instruction updates the LVDV table, including instructions from external libraries. However, if the code footprint causes too many replacements in the LVDV table we can optionally restrict the range of instructions which are allowed to access it, by excluding external libraries for example. In addition, to achieve the desired fault coverage, multiple experiments can be performed with different portions of the code being tracked, as suggested by Hangal et al. [20].

2.8. Experimental Methodology

To evaluate the effectiveness of our approach, we use four applications from the BugBench benchmark suite [27], *bc-1.06*, *ncompress-4.2.4*, *gzip-1.2.4* and *polymorph-0.4.0*, with a total of eight bugs. The applications that we use are representative, real applications with significant use in practice. The bugs in those applications are also real bugs rather than purposely injected ones. We were not able to test our approach on some of the other applications included in the BugBench suite because we were not able to compile or run those applications on our simulator.

In our experiments, we compare the hardware LVDV table to the software approach DIDUCE, in terms of bug-detection capabilities as well as number of generated false-positive alerts. To carry out the comparison, we performed two sets of experiments for each of the selected applications. In the first set of experiments, we used an infinite size LDVD table. The infinite size table tracks the addresses and values of memory operations, as well as the variance for all arithmetic instructions. With this idealistic setup, we mimic the software approach DIDUCE, where no hardware restrictions are imposed on the number of tracked expressions. In the second set of experiments, we used a single, realistic LVDV table with 4K entries 4-way set

associative, which only keeps track of addresses generated by store instructions. In both experiments, we used a single-bit precision variance (i.e. we did not use the chunks approach described in Section 2.1).

2.9. Experimental Results

In this section we use the four buggy applications to evaluate how our 4K LVDV table compares to DIDUCE. We also give a detailed analysis for some of the bugs and provide interesting insights about the strengths and limitations of our mechanism and DIDUCE. To facilitate discussion and to be able to contrast and compare our results, we grouped the bugs by their nature. The bugs in the first group are due to incorrect or missing bounds checking (of the loop bounds for example). Thus, a loop may execute too many times and either overflow or underflow a buffer. In the second group, the bugs are due to improper use of library calls, such as *sprintf* and *strcpy*.

2.9.1. Incorrect bounds checking

We first analyze two of the bugs from *bc-1.06*. BC is an arbitrary precision calculator language and it is also the largest application in our test suite with over 17000 lines of code. We trained our LVDV tables using several example programs such as computing prime numbers, square roots, etc. Then, we executed a specially crafted input program, which was able to trigger both bugs at the same time (this is possible, since in some cases the bug may corrupt an unused memory location and allow the program to continue executing successfully). The specially crafted input program was largely different from our training set and thus the LVDV tables signaled a large number of anomalies: 45 and 54 for DIDUCE and for the 4K LVDV

respectively (after eliminating duplicate anomalies with the same PC, and anomalies from external libraries). Thus, bc-1.06 exposed a general weakness in DIDUCE, as well as any other statistic rule-based approach: the quality of the reported results is related to the quality of the training set. However, even though the number of reported anomalies was large, those anomalies were clustered in several specific functions. Some of those anomalies were new-code anomalies, which indicated that these regions of code were rarely exercised. As noted by Hangal et al. [20], revealing such rarely executed code and corner cases is also useful to developers.

One of the bugs in *bc* is an interesting off-by-one bug as shown in Figure 6. The idea of the code is that whenever the *next_array* counter reaches the end of the *a_names* array, the function *more_arrays()* is called to increase the capacity of *a_names*. However, in this buggy code, the function *more_arrays()* is called one iteration too late and the array *a_names* is over-flown, as shown at line 4 in the figure. In other words, the correct condition should be "*if (next_array >= a_count)*" instead of "*if (id->a_name >= a_count)*".

In the assembly code of this program, a store word instruction is used at line 2 to overflow the array. Both the 4K LVDV and DIDUCE detected a larger than usual variance in the address of this store instruction and signaled an anomaly. In fact, two anomalies were signaled for the same store instruction: once, when the variance of the store address was increased from bit 6 to bit 7, and again when the variance was increased from bit 7 to bit 8. However, it is interesting to observe that such larger than usual address would be signaled even if we fixed the bug with the above suggestion. Therefore, both DIDUCE and the 4K LVDV do not literally detect this off-by-one bug, but rather they detect the unusually large address range of the store instruction. What makes DIDUCE or the 4K LVDV effective is that frequently such unusual behavior can point to the root cause of a real bug, as in this case.

```
1   id->a_name = next_array++;
2   a_names[id->a_name] = name;  /*detection*/
3   if (id->a_name < MAX_STORE){
4     if (id->a_name >= a_count){    /* bug */
5       more_arrays ();
6     }
7   return (-id->a_name);
8   }
```

Figure 6. An off-by-one bug in bc-1.06.

In the bug from Figure 7, the loop condition variable *v_count* is mistaken for a different variable *a_count*. Therefore, whenever *v_count* happens to be larger then *a_count*, the loop will continue executing and overflow the buffer arrays. Both 4K LVDV and DIDUCE detect the unusually large variance in the address of the store instruction writing to the buffer arrays.

```
/* Initialize the new elements. */
for (; indx < v_count; indx++){ /* bug*/
  arrays[indx] = NULL;  /*detection*/
}
```

Figure 7. Incorrect loop condition in bc-1.06.

For the benchmark *polymorph-0.4.0*, DIDUCE was very effective in detecting the defect, with no false-positives. The buggy part of the benchmark is shown in Figure 8. Polymorph is a filesystem "unixizer" [82]. It converts uppercase characters in a filename to lower case. It also removes unnecessary characters, such as "C:\\", which certain programs append to the beginning of filenames. The code in Figure 8 is from the function *convert_fileName* in polymorph.c. The for-loop iterates through all the characters in the original filename, converts them to lower case and stores them into the new filename: *newname*. However, if the original filename is longer than MAX, it can overflow the *newname* array and overwrite the stack return address. Originally, MAX was set to 2048. For ease of trigging the bug, we changed it to 64.

We trained the LVDV tables by running polymorph on several short filenames. After the training step, we provided a filename slightly longer than 64 characters and both 4K LVDV and DIDUCE signaled two anomalies. The first anomaly corresponds to the store byte instruction, which stores a character from array *original[i]* to array *newname[i]*. The second anomaly corresponds to the store byte instruction which appends the string terminating character '*\0*'to the array *newname[i]*. From this example, we can see that multiple anomaly alerts do not necessarily mean false positives since they may all point to the same bug.

```
char newname[MAX];
/* convert the filename */
for(i=0;i<strlen(original);i++){ /*bug*/
  if( isupper( original[i] ) ){
    newname[i]= tolower(original[i]);
    continue;
  }
  newname[i] = original[i]; /*detection*/
}
newname[i] = '\0'; /*detection*/
```

Figure 8. Buffer overflow in polymorph-0.4.0.

Lack of bounds checking causes the next bug in *ncompress-4.2.4*. The defect is in the *decompression* function as shown in Figure 9. The loop in Figure 9 performs no bounds checking and a carefully crafted input can underflow the variable *stackp*. The 4K entries LVDV table tracking store addresses was very effective in pointing out the exact defect location, with no false-positive alerts.

```
while((cmp_code_int)code >=(cmp_code_int)256)
{ /* Generate output characters in reverse order */
    *--stackp = tab_suffixof(code);  /*bug*/ /*detection*/
    code = tab_prefixof(code);
 }
```

Figure 9. Buffer underflow in ncompress-4.2.4.

2.9.2. Misuse of library functions, *sprint* and *strcpy*

The next four bugs are very similar in that they all misuse the library calls *sprintf* or *strcpy*. There was one such bug in each of the four evaluated applications. DIDUCE and our 4K LVDV were less effective in detecting those bugs as we elaborate next.

Due to the similarity of these bugs, we present an example of only one of them, in Figure 10. In the figure, *fileptr* corresponds to the filename of the input argument. A filename larger than *MAXPATHLEN* can overflow the *tempname* buffer and cause the stack return address to be overwritten. Neither 4K LVDV nor DIDUCE were able to directly identify this type of bug. However, for *gzip* both 4K LVDV and DIDUCE signaled anomalies originating from a function called "*name_too_long*". In addition to that, for the benchmarks *ncompress* and *gzip*, DIDUCE (but not 4K LVDV) signaled multiple anomalies to function calls *strlen( )* which computed the length of the input filename elsewhere in the code. Such anomalies provide a very helpful hint that the bugs are related to the length on the input string. Because our 4K LVDV monitored only store addresses, it did not produce the *strlen( )* anomalies. However, by allowing the flexibility to specify the types of instructions to monitor (such as arithmetic, or memory operations), the 4K LVDV would also output those helpful anomalies. Polymorph and BC, on the other hand, did not test the length of the input elsewhere in the code, and dynamic variance checking did not signal any helpful anomalies to track those bugs.

```
void comprexx(char **fileptr)
{ char tempname[MAXPATHLEN];
  strcpy(tempname,*fileptr); /*Bug String copy without
checking the length of the source and target buffers */
 }
```

Figure 10. Buffer overflow in ncompress-4.2.4.

DIDUCE as well as LVDV would be much more effective at pointing out the location of such a buffer overflow if there was an access of variables surrounding the buffer. Any overflow, which results in a high variance of those variables, would be easily detected by DIDUCE. This approach is similar to adding canaries to protect buffers. As part of our future work, we plan to use the compiler to insert load accesses to canaries at strategic locations in the code. These load accesses will then be automatically monitored by the LVDV table for enhanced buffer overflow protection.

In summary, we demonstrate that the limited size 4K LVDV successfully approximates the software approach DIDUCE. In particular, the 4K LVDV detected all four bugs which DIDUCE detected. Some helpful anomalies signaled by DIDUCE (variance in *strlen( )*) can also be signaled by 4K LVDV when it is allowed the flexibility to select the types of instructions to monitor (arithmetic or memory).

In terms of false-positive alerts, the 4K LVDV signaled a larger number (54 vs. 45) of anomalies only in the application *bc*, compared to DIDUCE. For the rest of the applications, the number of signaled anomalies was identical as shown in Table 2. The total number of signaled anomalies is shown after eliminating duplicate anomalies from the same instruction and anomalies from external libraries. Since *ncompress* requires different inputs to trigger the bugs, we provide the number of anomalies signaled for each input. For the rest of the benchmarks, a single input was sufficient to trigger all bugs.

Table 2. Total number of anomalies signaled by DIDUCE and 4K LVDV.

	polymorph	bc	ncompress (input 1)	ncompress (input 2)	gzip
DIDUCE	2	45	1	0	6
4K LVDV	2	54	1	0	6

2.10. Summary

In this chapter we realize that both soft-errors and software bugs manifest in similar ways during execution. We propose a unified approach to target both problems by exploiting a program locality called Limited Variance in Data Values (LVDV). We design a simple hardware structure to track instruction-level invariants and to detect abnormal execution behavior. In terms of soft error detection/recovery, our experimental results show that the proposed scheme significantly improves the Mean Time to Failure (MTTF) of both the issue queue and the functional units, by an average of 39% and 72% respectively. Negligible performance overheads are incurred for such reliability enhancements. For software bug detection, we demonstrate that our realistic LVDV mechanism is able to provide similar bug detection capabilities to the software tool DIDUCE while eliminating the performance overhead associated with software approaches, making it possible to monitor production code for bug detection.

CHAPTER 3. ANOMALY-BASED BUG PREDICTION, ISOLATION, AND VALIDATION: AN AUTOMATED APPROACH FOR SOFTWARE DEBUGGING

In this chapter, we differentiate between a software defect (commonly known as a bug), a defect infection and a program failure. We use the terminology from the book "Why Programs Fail" [73]. 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.

Software defects, present a serious challenge for computer system reliability and dependability. Once a program failure such as a program crash, an infinite 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][24][65](i.e., the process of determining all the instructions that have affected the failing instruction) is a tedious and time consuming effort, or if automated it 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. [76] proposed an algorithm to significantly reduce the slicing time and the storage requirements so as to make it practical. However, as pointed out in [17][75], even with efficient backward slicing, a nontrivial portion of the program needs to be examined manually to locate the faulty code.

Another promising technique to facilitate debugging is anomaly-based bug detection [14][15][20] as discussed in CHAPTER 2. 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 [20][77] 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 chapter, 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 11.

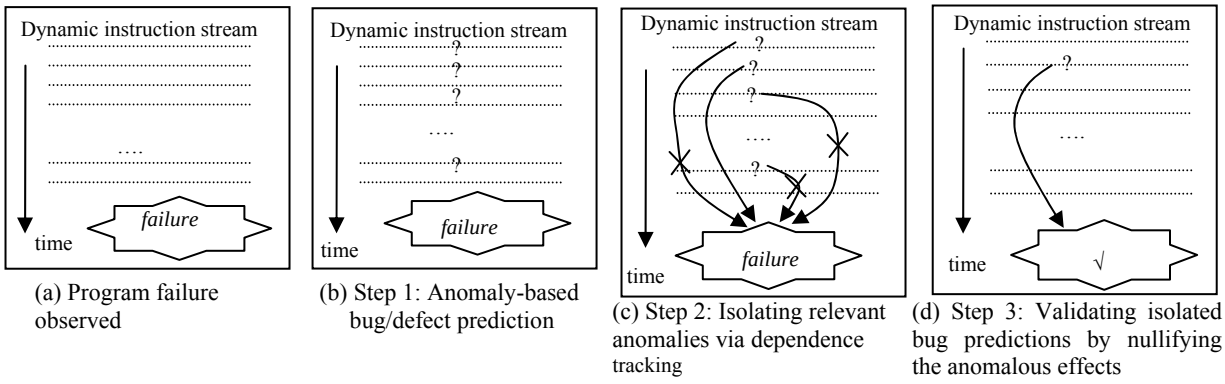

Figure 11. Overview of the proposed automated debugging process (the symbol '?' represents a predicted bug).

After a program failure is observed during execution (Figure 11a), 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 11b). In this work, we leverage two previously proposed bug detectors– DIDUCE [20] and AccMon [77], 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 3.5).

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 11c). 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 [10][11][53][61].

In step 3, we validate the isolated bugs by nullifying the anomalous execution results (Figure 11d). 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 chapter 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 using the PIN 2 dynamic binary instrumentation system [32], to emulate the proposed architecture support, which can also be used as a standalone software for automated debugging. We have made our tool available at http://csl.cs.ucf.edu/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 [17] our approach locates the defects more accurately and presents the user with a much smaller code set to analyze.

3.1. Predicting Software Bugs

3.1.1. Method

Previous research [14][15][20] 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 [20][77] 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 chapter, we leverage two previously

proposed anomaly detectors DIDUCE [20] and AccMon [77], as detailed in Section 2.2.2 and propose a new one based on loop-count invariance.

In general, program defects may result in abnormal control flow behavior and branch mispredictions can be used to detect control flow anomalies. One possible approach is to use mispredictions of branches with high confidence, as exploited by Wang et al. [63] 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 chapter, we propose to focus on one special type of branch, loop branches. We learn the normal range of loop iterations during passing runs, and detect anomalies such as too few or too many iterations 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 3.2. We call this bug predictor LoopCount. As we will show in our experimental results (see Section 3.5), such a simple loop-based bug predictor is effective in catching some interesting memory corruption defects, which DIDUCE misses.

Next, we revisit in more detail the code example from bc-1.06 from CHAPTER 1 to illustrate each of the bug predictors. Figure 12 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. */
18    for (; indx < v_count; indx++){  /* defect: incorrect loop condition */
19      arrays[indx] = NULL;     /* infection: overflows its size information */
20
21    /* Free the old elements. */
22    if (old_count != 0){
23      free (old_ary);    /* crash: when the buffer size is corrupted */
24    }
25  }
```

Figure 12. 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 12, 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. The anomaly on line 19 is the immediate infection point of the defect on line 18 and thus 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 12, 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 by Zhou et al. [77] 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 that 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 of AccMon we choose to monitor every memory update and use automated bug isolation to eliminate false positives (see Section 3.2).

3.1.2. 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 as described in CHAPTER 2. As highlighted CHAPTER 2, 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 resources, 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 3.5, 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.

3.2. Isolating Relevant Bug Predictions

3.2.1. Method

As discussed in Section 3.1, 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.

3.2.2. Architectural Support

In this chapter, we propose to construct the dynamic forward slices in hardware by leveraging tagged architectures proposed for information flow tracking or taint tracking [10][11][61]. 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 12. 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 [72][74]. The delta debugging algorithm is a divide-and-conquer approach, which is used to automatically simplify and isolate failure inducing input [74], failure inducing differences in program state [8], as well as failure inducing cause-effect chains [71]. Conceptually, our anomaly isolation algorithm works as follows. First we divide the anomalies in half, and allow 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 13 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 13 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 $n^2 + 3n$ [74], 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.

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 13. 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.

3.3. 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 CHAPTER 3, 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 as before**. 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 fails differently**. 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.

3.4. Experimental Methodology

3.4.1. 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 [32]. 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 3.1, 3.2 and 3.3, 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.

3.4.2. 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 3. Six of the applications are from the BugBench suite [27]. Some of them contain more defects than those shown in Table 3, 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 two orders of magnitude more lines of code than any of the BugBench programs. The real bug in *gcc* is analyzed in [71].

Table 3. 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

3.5. Bug Detection Results

Table 4 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 3.3 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 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 4 are obtained using large 64K-entry prediction tables. The impact of hardware implementation and limited table sizes is discussed in Section 3.5.2.

We can make several important observations from the results in Table 4. 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 3.3, 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 4, 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.

Table 4. 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**

3.5.1. 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 14.

The root-cause of the failure is a function call to *apply_distributive_law* in combine.c: lines 4013-4018, listed in Figure 15. 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 16 (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 16 (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 14. 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 15. GCC defect: the call to apply_distributive_law creates a cycle in the RTL tree.

Zeller [71] 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 15. 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 16 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 16 (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 do 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.

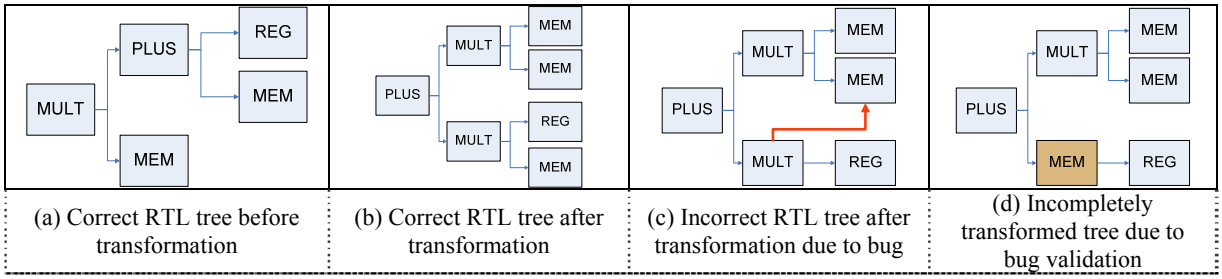

Figure 16. GCC RTL tree transformations before and after function call to “apply_distributive_law”.

3.5.2. 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 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 3.1.2 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 5. Compared to Table 4, 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.

Table 5. 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**

3.5.3. Comparison to Other Approaches

In this section, we compare our proposed approach to a state-of-art debugging technique based on failure-inducing chops [17]. In this technique, the minimum failure-inducing inputs are isolated using delta-debugging [74]. 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 6, 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 6. 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

3.6. 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 [66]. 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 the book "Find the Bug" [5], 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 [57] may result in a safer program state.

Fourth, this chapter 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.

3.7. Related Work

In this section, we highlight the limitations of our proposed automated debugging approach. 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 [14][15] to facilitate program evolution and detect software defects. DIDUCE [20] and AccMon [77], as described in Section 3.1, exploit a compact representation of value-based or store-set invariants. Program anomalies have been shown useful to detect inconsistent use of locks [54] or atomicity violations [28][31] in multithreaded programs. In [49], dynamic invariants have also been used in detecting and filtering soft errors.

Code coverage or spectra between passing and failing runs [21][50] 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 [62][65] 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][24] 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 [75] 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 [17], which we discuss in Section 3.5.3.

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 [72]. Subsequently, delta debugging is used to isolate and simplify failure-inducing input [74], to isolate failure inducing differences in program state [71], and to obtain cause-effect chains [8] 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 [34] can also be used to improve our bug isolation process.

Nullifying Instructions Concurrently to our work, D. Jeffrey et al. [23] 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 [78] 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 [22][35][39][48][60][67][68] 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.

3.8. Summary

In this chapter, 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 in the case of the *gcc* bug, our approach was able to eliminate the huge majority of false-positives and still remain effective even for such a large software program.

CHAPTER 4. TIME-ORDERED EVENT TRACES: A NEW DEBUGGING PRIMITIVE FOR CONCURRENCY BUGS

Debugging shared-memory parallel programs is notoriously hard due to the inherent non-determinism present in these programs. The non-determinism stems from the fact that multiple threads may be selected for execution and running simultaneously on different processor cores or alternating on a single core as scheduled by the operating system. Usually these threads are not independent and they cooperate in performing certain functions and sharing resources. Concurrently executing threads may potentially interleave their accesses to shared resources in arbitrary fashion. If the software developer fails to anticipate all the possible thread interleavings, she opens the door for possible erroneous behavior.

Since concurrency bugs are usually caused by such unforeseen thread interleavings, they may be very difficult to reason about. In addition, since these bugs may manifest only under certain thread interleaving they may be very difficult to reproduce reliably, causing only sporadic failures. The difficulty in reproducing those bugs also makes the use of traditional cyclic debugging techniques, such as breakpoint debugging, more difficult.

In this chapter, we propose a debug primitive to facilitate debugging parallel programs by making non-determinism visible to the programmer in the form of a time-ordered trace of events. The key idea is to assign a global time-stamp to different events of interest. The architectural support for the proposed primitive is lightweight. We envision three ways to utilize the proposed debugging primitive. First, by letting the recording of time-ordered events be always ON, we buffer the last N function calls/returns so as to reconstruct thread interleaving at the function level right before a failure point. The motivation is that when debugging a program, the

programmer is usually not interested in the execution history of the entire program. Instead, she is interested in answers to specific questions related to a failure, such as: "Which threads were actively executing just before the crash, and how were they interleaved?" In a sense, the proposed primitive provides a way for post mortem analysis, like a core dump or stack trace for parallel program execution. Our case studies show that the time-ordered trace based on the last N function call/return is typically most effective among the three ways of utilizing the primitive to reason about the root causes.

Second, through a software interface, the programmer can direct the primitive to monitor function interleaving in a specific region of code. This mechanism seeks to answer questions like "which functions were executing concurrently to function *foo(),* when the incorrect results were produced?".

Third, the programmer can direct the primitive to monitor interleaved accesses to shared variables. This mechanism serves similar goals to the existing 'watch' primitive iWatcher [78] in a debugger but extends it to be a 'parallel' watch. Such a watch can also be used together with function call/return monitoring.

Compared to existing work on concurrency bug detection, the goal of our proposed scheme is not to provide an automated approach to detecting a specific type of bugs. Instead, it is a generic primitive, which we believe is helpful in a wide variety of scenarios. Compared to record/replay for reproducing concurrency bugs, our approach is much more light-weight, exposing non-deterministic events only for a limited scope specified by the programmer. Our focus towards light-weight, allows our approach to be always-ON and minimally perturb the program execution.

4.1. Current State of the Art

Given the difficulty of debugging concurrency bugs, various techniques have been proposed in previous work. Many of those techniques have focused on automatically detecting data races [7][9][42][46][47][70][54] and more recently atomicity violations [16][28][69]. Unfortunately, data-race and atomicity detection techniques usually focus on the synchronization of a single (or several [30]) variable(s) and fail in more complex scenarios where the bug involves complex data structures, or the file system. Moreover, while these techniques are helpful for some bugs, our experiments show that there are many more bugs which remain unaddressed – for instance complex atomicity bugs, order violation bugs and logical concurrency bugs. We discuss these in more detail in our case studies in Section 4.4.

Static [13] and dynamic [4] deadlock detection techniques have also been proposed. While static approaches are very valuable, they require program annotations and may potentially cause a large number of false-positives. This would be especially true in MySQL, where many deadlocks are not bugs and are handled automatically by the innodb engine. On the other hand, if a deadlock bug has already slipped into production, dynamic approaches incur substantial performance overheads and thus are not suitable for always-ON or production runs.

Another direction aims towards reproducing concurrency bugs, by utilizing record and replay tools, either purely in software [2][25][43][45] or with hardware assistance [22][36][35][39][40][68][67]. The purpose of these tools is to capture the sources of non-determinism in a multi-threaded program (e.g. the order of accesses to shared memory) so that when a bug is triggered, the program may be replayed with the same thread interleaving. Record/replay schemes enable cyclic debugging for parallel programs since the bug may be

reproduced every time. Recording schemes may also provide the illusion of debugging backwards in time [6]. On the downside, these mechanisms either incur high performance overheads or require significant hardware changes. Moreover, large trace files, which grow at a high data rate, are typically required for record and replay. Such overheads may be prohibitive for always-ON use, especially in scenarios where the bug is very rarely triggered. The performance overheads, in addition, can perturb the program execution so much, that the bug is no longer triggered. As previously discussed, our proposed primitive is different from the record/replay schemes. It, however, can benefit from deterministic replay. For example, in a debugger with replay capability, the primitive can be used to monitor different code regions in different reruns. On the other hand, our primitive can be a standalone mechanism and is not dependent upon deterministic replay.

Generating debug traces is not a new idea, and 'printf()' is arguably the most commonly used way for such a purpose. Many commercial software products are also able to produce a trace if compiled with a debug build. For instance the MySQL server is capable of producing a very detailed trace of function calls/returns, indented by the call depth and marked with the thread ID of the executing thread. In this chapter, we revisit this fundamental concept of creating traces, in the context of multi-threaded programs. We believe that *efficiently* creating a trace is even more valuable in the context of concurrent programs. Unfortunately, creating traces completely in software, as in MySQL incurs significant performance overheads (about 100X in our experiments), which prevents the use of such traces for long periods of time, or always-ON. Moreover, software only traces are not able to reveal fine-grain time ordering of events since trace collection (e.g., with 'printf()') may perturb thread interleaving.

Virtual machines, such as the Java VM have also been enhanced with the capabilities to record traces. The tracing facility in Java is also able to time stamp method calls with microsecond precision [81]. Still, there is no certainty that this precision is enough and performance may still be significantly impacted.

More recently, Nagarajan et al. [38] proposed to expose cache coherence events in a multicore system to the software. The proposed mechanism was used to support speculation and record-replay systems. However we believe that it is too low-level to be used directly by the programmer for debugging.

4.2. Time-Ordered Event Traces

Our proposed approach consists of a software interface that allows the programmer to communicate with the trace collection engine, and a hardware component which facilitates the efficient collection of time-ordered event traces. In the following section we elaborate on our design.

4.2.1. Software Interface

For the purpose of debugging parallel programs, we found that the program events likely to be most useful to the programmer were function call/return and memory read/write. Thus, in this chapter we focus on collecting only those events. However, our approach can easily be extended to collect other events as well (such as branch outcomes or branch mispredictions, cache misses, etc.) if they are deemed useful. By default our approach collects only function call/return events but we allow the programmer to specify memory region(s) for collection of memory read/write events as well.

We envision three modes of operation. In the first case, the programmer is interested in the sequence of events which have occurred just before a program failure, such as a crash, incorrect results, etc. In this case, trace collection is always-ON, buffering the last N events in each thread, where N is fixed. We call this mode FIFO mode, since the event buffer is organized as a FIFO queue, with new events overwriting the old ones. By default, we assume that the number of collected events N is equal to 2K loads/stores, or about 4K calls/returns (since we are able to combine calls and returns, Section 4.2.2). We chose this number N, since it has been sufficient in all the debugging scenarios that we have examined (see Section 4.4).

In the second and third modes, trace collection is controlled explicitly by the programmer. The second mode is used to collect function call/return traces within a user specified region while the third is a parallel watch of memory variables. The explicit modes are most useful, when the programmer wants to inspect a certain region of program execution such as the thread interleaving during a function call, or a sequence of events triggered by a button click on a graphics user interface (GUI). Explicit modes are controlled by a set of API calls from within the program or by using system calls to toggle trace collection from outside of the program, e.g. using an external debugging tool such as GDB. In an explicit mode, the number of events collected depends on the size of the region specified by the programmer. New events do not replace old ones, but keep accumulating in the trace. If the trace becomes too large, it may result in performance degradation as the trace needs to be written to memory (see Section 4.2.2).

To support the envisioned modes of operation, the following API calls are designed to communicate with the trace collection engine. These APIs may be invoked from within the application, or externally. In some cases it is more convenient to enable/disable tracing externally, while running or debugging the program. For instance if the program enters a

deadlock, we would like to dump the last N event at that point. Or if we know that the bug may be triggered by some external event, such as pressing on a button in the browser, we might want to enable tracing just before that point without modifying the binary.

- *trace_start(call_depth, scope).* This API call is used to start collecting a trace of function call/return events. The optional call depth parameter is useful, if we want to avoid collecting events for some un-interesting low-level function calls. Function calls/returns greater than *call_depth* will not be collected. The *scope* parameter enables us to specify whether trace collection should terminate once the function calling *trace_start()* returns (thus we do not have to guard on all the possible function exits with a *trace_stop()*). When trace collection is enabled, it is enabled for all threads since we want to capture the thread interleaving. If multiple threads call *trace_start(),* trace collection terminates when all of the threads call *trace_stop()* or they all exit out of the scope of the function calling *trace_start().*

- *trace_mem_start(addr_low, addr_high).* This API call enables the programmer to specify a memory region for collecting load/store events across all threads. It is similarly terminated by *trace_mem_stop()* or when it leaves the scope of the calling function.

- *trace_dump(output_file)* – dump the collected traces to a file, regardless of the collection mode.

- In addition, we provide API calls to set the trace mode, FIFO or explicit, to set the call_depth or the number of events collected (N) in explicit modes.

4.2.2. Architectural Design

The main goals of using architectural support to construct the traces are: fine-grain time-stamping of events and performance efficiency (so that we do not perturb the original program

execution). The architectural support should not require significant architecture changes and should be light weight. In this section, we elaborate on how we propose to achieve those goals. We limit our discussion to multi-core and multi-processor shared-memory systems. We leave distributed shared memory systems (clusters) and message passing systems as part of our future work.

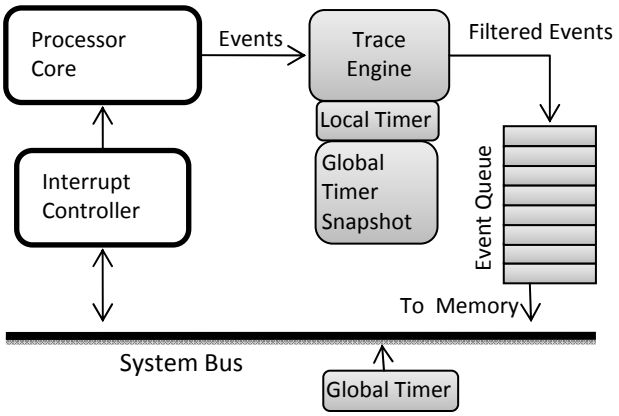

Figure 17. Proposed architectural support with new components colored in gray.

Time-Stamping Events. In our proposed approach, each thread logs its events into a local trace and uses a local timer to time-stamp events in its trace. The local traces of each thread are then merged to construct the final global time-ordered trace. The benefits of such approach are scalability and performance efficiency, since threads do not have to content for a single resource such as a global timer. However, the challenge is that all the local timers must be synchronized to a common source, so that the global time can be reconstructed in the final trace. Also, the local timers must remain consistent in the presence of context switches, thread migration and different frequency settings of each processor core.

Fortunately, global synchronization across processors and core-frequency independent timers are not unique requirements to our approach. In fact, similar hardware support is already present in commercial systems. Most current x86 processors contain a component called Advanced Programmable Interrupt Controller (APIC) [80]. The APIC controller is responsible for accepting and generating interrupts. The APIC controller is also able to forward interrupts to remote processors in the system, using 3 dedicated wires (in the Pentium 6 family processors) or using the system bus (in Pentium 4 and Xeon family processors). Once an interrupt is delivered to APIC, then APIC may forward this interrupt to the processor core, which invokes the interrupt handler. In a multi-processor setting, APIC is useful in a variety of scenarios [51]. In one use-case, we may have two threads of the same process concurrently executing on different CPUs. Thread_1 may unmap a memory region and mark the corresponding page table entries as 'not-present'. CPU_1 must then wipe-out any remaining cache-lines and TLB entries corresponding to that memory region. At the same time, CPU_1 must inform CPU_2 of the unmap operation and force it to invalidate its cache-lines and TLB entries as well. This is achieved by using a high priority interrupt signal delivered to the remote APIC of CPU_2. As another example, a debug breakpoint hit by one thread, must stop the execution of all threads in the process (which is the default behavior of GDB [58].) This is achieved by having the thread which first reached the breakpoint to trap into the interrupt handler and distribute an interrupt to all other threads running on different cores.

Another interesting feature of current APICs is that they have a 32 bit count-down counter/timer, which operates at a constant frequency, independent of the processor P-states or frequency [80]. This is achieved by using the system bus clock, instead of the processor core

clock. The count-down timer may be used for generating a self-interrupt, for preemptive scheduling purposes, or timing the performance of certain operations.

In our work, we leverage the presence of such architectural support in the system in order to maintain consistent time-stamps across threads, and to support the API presented in Section 4.2.1. The architectural support that we propose is illustrated in Figure 17, with the new components added by our approach colored in gray. The new components consist of one global timer connected to the system bus (if one such timer is not already present in the system), a per-core trace engine and a per-core event queue. The global timer is driven by the system bus clock and thus runs at a constant frequency. Whenever a thread is scheduled for execution by the OS, the thread reads the global timer and stores a snapshot in a local register in the trace engine, as shown in Figure 17. At the same time, the thread resets its local timer to 0 and starts using it to time-stamp events. When a trace is dumped to file using an API *trace_dump()* call, or when the thread is context switched out, the snapshot of the global timer is stored into the trace. This way, the global time can be easily reconstructed by adding the local timer to the snapshot of the global timer. We assume that acquiring of the snapshot of the global timer takes constant amount of time, using the system bus, or by using dedicated wires. Even in the event that acquiring the global timer does not take constant time, we believe that the error will be very small and not critical, since such synchronizations are rare – only on context switch. The local timers may run at a constant frequency, driven by the bus clock, or they may scale with the core frequency. In this chapter, we assume that the local counters run with the core frequency. The benefit of this approach is that it gives us more fine-grain timing of events, since the core clock runs faster than the system bus clock. Whenever the frequency of the processor core changes, we store a frequency change event in the trace and reset the local counter. An alternative approach to

synchronizing the local timers would be to synchronize them to each other (by using the interrupt lines) instead of to the global timer. While we believe that either approach is feasible, we select to use a global counter due to its simplicity.

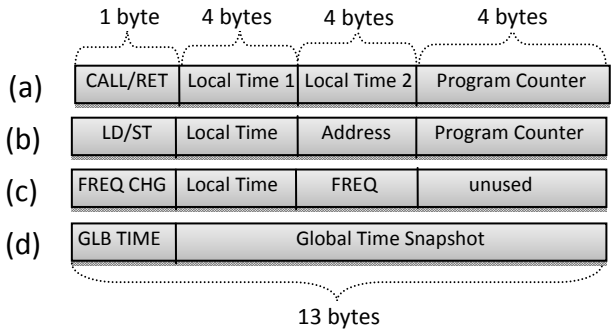

Figure 18. Time-ordered trace entry format. (a) combined call/return event (b) memory load/store event (c) frequency change event (d) global timer snapshot.

Trace Collection. The trace engine in Figure 17 receives events (function calls/returns and load/stores) as they retire from the processor core. The trace engine filters those events and decides if they should be pushed into the trace. For instance, if the user has specified memory region(s) for monitoring, only those load/store instructions which have accessed the memory region will be included in the trace. The trace engine also keeps track of the current call depth of function calls/returns and includes them in the trace if their call depth is not greater than the one specified by the user. Keeping track of the call depth is also useful to detect when a function calling the API *trace_start()* is returning, in which case we may need to stop tracing, as described in Section 4.2.1.

After the events have been filtered by the trace engine, they are pushed into an on-chip queue, called Event Queue. In FIFO mode the event queue is managed as a circular buffer and maintains the last N events. In the explicit modes, new events do not overwrite the old ones, and

the event queue is simply used for buffering, so that old events can be written back to memory when the bus is available. In addition to buffering events, the event queue serves another useful purpose, which is to combine events and compress the footprint of the trace. For instance, if a function return event arrives, which matches a function call in the event queue, then the two events may be combined, as shown in Figure 18 (a). The first time stamp entry *Local Time 1* corresponds to the *call* and the second time stamp *Local Time 2* corresponds to the *return*. If the function call/return cannot be combined, then the second time slot is unused. In the common case, function calls/returns are combined, essentially reducing the storage requirements of the trace in half. Note that we do not need to perform a sequential search of the queue in order to combine call/return events. The trace engine keeps a small stack of pointers, pointing to the last function call at each call-depth in the queue. When a return arrives at a given call depth it is directly forwarded to the event queue using the pointer. Load/store events may not be combined, however we reuse the second time slot in the trace entry to store the load/store address Figure 18 (b). Additional event types stored in the trace are frequency change event and global timer snapshot, as discussed in Section 4.2.2 *Time-Stamping Events* and as shown in Figure 18 (c) and (d) respectively. From the figure we can see that each event occupies 13 bytes. Thus, the storage requirement of the event queue to maintain the last 2K events is 2k * 13 = 26kBytes per core. Due to combining of call/return events, this gives us about 4K call/return history.

API Support and Performance. The main goals of using architectural support to construct the traces are: fine-grain time-stamping of events and performance. To support the API calls described in Section 4.2.1, which control trace collection, we utilize the existing interrupt support. In particular, on a call to *trace_start()/ trace_mem_start(),* the thread triggering the event generates an interrupt to all other threads in this process. This is a similar mechanism to the

one used for breakpoint in multi-threaded programs. The difference is that the cores receiving the signal simply turn ON the trace engine and program execution is actually not interrupted. The calls to *trace_stop()/trace_mem_stop()* are handled similarly. On a *trace_dump()* the event queues of each core are drained to memory and written to a file.

The event queue must also be drained to memory whenever a thread is context switched out. But we do not need to restore the event queue when a thread is switched back in. Therefore, in FIFO mode we only incur performance overhead on a context switch, due to reading of the global time stamp and draining the event queue to memory. Since context switch is a rare event, this performance overhead should be minimal. On the other hand, in explicit modes, the event queue may not be large enough to hold all the events specified by the user. In this case, writing the events back to memory will consume bus bandwidth and result in performance degradation. The bandwidth consumed will depend on the events that the user decided to monitor (load/store or call/return) as well as the level of detail (call-depth of call/return events). Thus, the programmer should use explicit modes when the monitored region is small and fits in the event queue, or when the performance perturbation does not prevent reproducing the bug, possibly in combination with a record/replay tool.

4.3. Experimental Methodology

In order to evaluate our ideas, we implemented a prototype time-ordered tracing tool using dynamic binary instrumentation with Pin [32]. Using our tool, we are able to execute and trace unmodified x86 binaries on Linux. If the program executes one of the API calls (*trace_start, trace_stop, trace_dump*, etc.) as inserted by the programmer, our tool detects the API calls and enables/disables or dumps the trace to file appropriately. We also implement a

mechanism to control the trace generation/dump externally by sending signals to the debugged program. Our Pin-tool intercepts the signal and dumps the trace or enables/disables tracing. Due to the performance overhead imposed by our binary instrumentation, some of the bugs were much more difficult to trigger. Thus we inserted *sleep()* or *printf()* when necessary to reproduce the bug. We plan to release our Pin implementation as well as all supporting scripts and tools for processing a time-ordered trace at: http://csl.cs.ucf.edu/debugging.

Where appropriate in the case studies, we also compare our approach to the automatic atomicity-violation detection tool AVIO [28]. AVIO works by monitoring the cache-coherence traffic and detecting unserializable interleaving to a variable. Those interleavings, which do not occur during passing program runs but only occur during a failing run, are reported to the developer. Since we do not have the extra ISA instructions to specify the I-instruction and the P-instruction used in AVIO, we slightly modified the implementation of AVIO while preserving its functionality. In our implementation, we keep track of two coherence states per word: the current coherence state and the "previous" coherence state. Whenever the current coherence state changes due to a remote access (invalidate or downgrade), we save it as the "previous" coherence state. On a subsequent access by the local thread, we check the previous coherence state, the current coherence state and the current access type and we determine if an unserializable interleaving has occurred.

4.4. Experimental Results

In this section we present five case studies, one for each of the bug types: deadlock, atomicity violation, order violation and logical bugs. We also present a case study with bugs where our approach is *not* helpful.

4.4.1. Case study: Deadlock

Deadlocks occur when threads are involved in a circular wait for resources. Deadlock is one of the most common types of concurrency defects, estimated to about 30% of all concurrency defects, according to a recent study [29]. Our experience with browsing the bug databases of MySQL and Mozilla, confirms those findings. In the following case study we present a detailed example of how time-ordered traces help to reveal a deadlock bug.

The deadlock that we present in this case study appears in the rare case, when two users connected to a MySQL server concurrently issue account management commands to the server. The account management commands may involve setting/changing passwords or permissions for databases. For instance, if one user issues the *GRANT* command, while at the same time another user issues the *FLUSH PRIVILEGES* command, MySQL may deadlock.

To debug this problem, the programmer may dump a time-ordered trace containing the last N events leading up to the deadlock. A simplified version of the trace is presented in Figure 19. From the figure we can easily see that threads 1 and 2 are involved in a circular wait for two resources: the *acl_cache* lock and a table lock. More importantly, from the time-ordered trace in Figure 19, we can see exactly how the threads interleaved in order to reach this state. Note that this information is not available in traditional stack dumps, as we can see from Figure 20. The traditional stack dump only reveals the last resource that each thread attempted to acquire. It does not reveal all the functions involved and their interleaving. This is important information, since in large software acquiring of locks may be nested deep in different functions. In fact, since we were not familiar with the MySQL code, we fully understood this deadlock only after we obtained the time-ordered traces.

	Thread 1: FLUSH PRIVILEGES	Thread 2: GRANT
1		< mysql_grant()
2		. < simple_open_n_lock_tables()
3	< acl_reload()	
4	. < mutex_lock(&acl_cache->lock)	
5	. > mutex_lock(&acl_cache->lock)	
6	. < acl_init()	
7	. . < simple_open_n_lock_tables()	
8		. . < lock_tables()
9		. . > lock_tables()
10		. > simple_open_n_lock_tables()
11		. < mutex_lock(&acl_cache->lock)
12	. . . < lock_tables() //deadlock	// deadlock

Figure 19. Time-ordered trace of MySQL bug 12423. The first column shows the global time; next two columns show the function interleavings of the involved threads. The notation '<' means a function call, '>' means a return. The notation '...' represents the call-depth.

Thread 1: FLUSH PRIVILEGES	Thread 2: GRANT
< acl_reload() . < acl_init() . . < simple_open_n_lock_tables() . . . < lock_tables() //deadlock	< mysql_grant() . < mutex_lock(&acl_cache->lock) // deadlock

Figure 20. Stack dump of MySQL bug 12423. The stack dump alone does not reveal the deadlock interleaving leading to a deadlock.

Another deadlock bug that we investigated is MySQL bug 29154. This case is very interesting, since the innodb database engine for MySQL is able to detect and recover from deadlocks automatically. However, the problem occurs when, user_1 connected to a MySQL server sends a request to lock a number of database tables. The request may succeed for some tables, but then fail for other tables. MySQL detects this deadlock condition after a timeout, and aborts the transaction. The problem is that while aborting the transaction, MySQL does not properly release all the locks that user_1 has obtained. Subsequent transactions will also fail and

get aborted, since they require the lock still held by user_1. Similar to our example in Figure 3, time-ordered traces are very helpful to reveal how the deadlock occurs. However, the deadlock in bug 29154 does not involve fine-grain function interleaving, and thus a regular trace (not time-ordered) could be equally helpful in this case.

4.4.2. Case Study: Atomicity Violation

Another very common type of concurrency defect is an atomicity violation, accounting for about 30% of concurrency defects, according to Lu et al. [29]. Atomicity violations are caused when a section of the code is assumed to be atomic, however it is not properly guarded by synchronization and a remote thread may interfere. In the following case study, we also compare to the automated atomicity detection tool AVIO.

An example of atomicity violation is presented in Figure 21. In this MySQL bug, the binary log is temporary being closed, so that logging can continue to a different file. Such log rotation is usually performed, when the old log file becomes too large, or when a user explicitly issues a FLUSH LOGS command. Unfortunately, the developers did not realize that the closing of the old log file and the opening of the new log file must be performed atomically. In this bug, an SQL *INSERT* operation issued by a different thread, does not reach the binary log, since it thinks that the log is closed, a state exposed by the atomicity violation.

```
new_file() {
  save_log_type=log_type; // log_type is LOG_BIN
  close() // close old log file.
    ↘ log_type = LOG_CLOSED;
                <---------- remote insert in sql_insert()
                            if(log_type != LOG_CLOSED)
                               mysql_bin_log.write()   // not executed
  open(save_log_type) //open the new log file.
     ↘ log_type = LOG_BIN;
```

Figure 21. MySQL bug 791. An insert operation is not being recorded into the binary log.

To start debugging this problem, we first look at the failure symptom - binary log missing an *INSERT* operation. Based on this failure, one hypothesis that the developer may have is that the *mysql_bin_log.write()* function did not execute inside function *sql_insert(),* highlighted in italics in the figure. To validate this hypothesis, and to understand what may cause this to happen, we enabled always-ON trace collection, buffering the last N events. We also inserted an API call to dump the time-ordered trace if the log appears to be closed. In other words, the trace is only printed out when the bug is triggered. In addition, to understand which other threads or functions modify variable *log_type*, we enabled monitoring of that memory address. The time ordered trace that we obtained is presented in Figure 22. Our trace helps the developer find out which thread is causing the binary log to appear closed and why.

	Thread 9: FLUSH LOGS	Thread 10: INSERT
1	< new_file()	
2	. save_log_type=**log_type**;	
3	. < close()	
4	. . **log_type** = LOG_CLOSED;	
5	. > close()	
6		< mysql_insert()
7		. < open_and_lock_tables()
8		. > open_and_lock_tables()
9		. **log_type** != LOG_CLOSED

Figure 22. Time-ordered trace of MySQL bug 791.

In our experiments, this bug was also easily detected by AVIO, which recognized the *Write (remote Read) Write* interleaving as described in Figure 21. AVIO is well suited to detect bugs involving only a single variable, such as this. However, as we show in our next example, some atomicity violations escape AVIO, since they involve more complex data structures or the file system.

The atomicity violation bug that we present in our next example occurs during concurrent execution of commands *DROP TABLE* and *SHOW TABLE STATUS*, submitted to the MySQL server. The outcome of the bug failure is that the command *SHOW TABLE STATUS* fails with an error message about an un-existing table. As we can see from Figure 23, function *get_all_tables()* (implementing command *SHOW TABLE STATUS*) consists of two main logical components. First, it scans the file system and creates a list of tables*, make_table_list().* Second, based on that list, it opens each table and displays the required status information, *open_normal_and_derived_tables().* Creation of the table list and displaying the status information need to be performed atomically, since the table list may change between the two operations, e.g. a table may be dropped. Interestingly, the bug is not fixed by enforcing atomicity, since it would severely limit concurrency. Instead, atomicity violations are allowed to happen, but are later detected and handled correctly.

Since command *SHOW TABLE STATUS* results in an error message, our debugging strategy is to print a time-ordered trace containing the last N function calls/returns leading up to the error message. Once the error is triggered, we examine the trace, which is shown in Figure 23. Interestingly, from the trace we can see that function *my_delete()* in thread 1 runs simultaneously with *make_table_list()* in thread 2, highlighted in italics. However, the table is

added to the list just before it is deleted from the file system. Subsequently, function *open_normal_and_der_tables()* in thread 2 fails and we continue to print an error message.

AVIO is not able to catch this atomicity violation, since the race is to the file system and no variables were shared in memory; one thread is deleting a file while another thread is scanning the directory and reading files. In addition to this bug, we experimented with two other atomicity violation bugs involving the file system – MySQL bugs 2385 and 2387. Both of those bugs involve concurrently executing commands such as *CREATE TABLE* or *ALTER TABLE*. In both cases the atomicity violation results in a corrupted or overwritten table definition file (.frm). By creating a time-ordered trace during the execution of these commands, the bugs are clearly exposed with our approach. On the other hand, these bugs cannot be detected by AVIO. Notice that even though *CREATE* and *ALTER* may manipulate the same table, AVIO is not able to detect the bug, since when the threads open a table using *open_table()* they get allocated a new table object populated from the table definition file (.frm). Thus there is no data race on the table object.

	Thread 1 DROP TABLE	Thread 2 SHOW TABLE STATUS
1		< get_all_tables()
2	< mysql_rm_table()	
3	. < mysql_rm_table_part2()	
4	. . < ***my_delete()***	
5		. < ***make_table_list()***
6		. . < ***add_table_to_list()***
7	. . > ***my_delete()***	
8		. . > ***add_table_to_list()***
9		. > ***make_table_list()***
10		. < open_normal_and_der_tables()
11	. . < Query_cache_invalidate()	
12	. . > Query_cache_invalidate()	
13		. . < open_table()
14	. > mysql_rm_table_part2()	
15		. . . < openfrm()
16	> mysql_rm_table()	
		. . . . < my_error() // error displayed

Figure 23. MySQL bug 27499. Command DROP TABLE may race with command SHOW TABLE STATUS.

4.4.3. Case Study: Order Violation

Order violation bugs are caused when the desired order of two operations performed by different threads is flipped. Lu et al. [29] reported a large number of order violation bugs (15 order violations out of 41 studied bugs) in Mozilla. Order violation bugs are particularly difficult to debug and to fix, as we show in the following case study. Moreover, as pointed recently [29], order violation bugs have received very little attention in the research community.

Consider the order violation bug in Mozilla shown in Figure 24. This bug is present in the java script engine in Mozilla. After a java script thread has finished execution, it calls function *js_DestroyContext()* to cleanup and remove its context from the list of active contexts. In case the thread is the last one entering *js_DestroyContext()* it also performs more extensive garbage collection and de-allocates storage associated with the java script runtime. Unfortunately, under a

certain rare thread interleaving, a next-to-last thread may take a very long time to execute in *js_DestroyContext().* In the meantime, the last thread to enter the function advances faster and de-allocates storage still in use by the previous thread, causing a crash.

This bug cannot be fixed by simply adding locks. In addition, this bug has proven to be very difficult to reason about (we have presented only a simplified version for clarity), since it has been fixed multiple times and reappeared in different forms for the last 7 years!

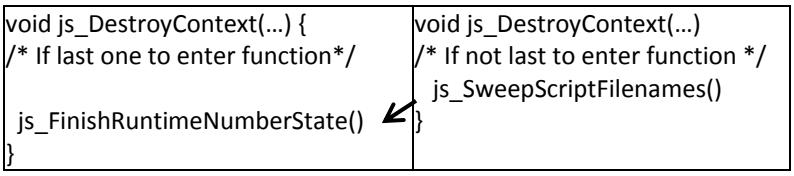

Figure 24. Mozilla bug 515403. Function *js_FinishRuntimeNumberState()* deallocates a hash table that *js_SweepAtomState()* later uses. The correct order is specified with an arrow.

Since the bug results in a crash, a natural debugging strategy using our approach is to dump a time-ordered trace at the time of the crash, shown in Figure 25. From the trace, we can observe that the two threads enter *js_DestroyContext()* almost simultaneously, however thread 2 appears to be the last one entering the function since it calls *js_FinishRuntimeNumberState()* (only the last thread calls this function) followed by garbage collection in function *js_GC().* In the meantime, thread 1 has taken a long time to execute, and crashes during garbage collection in *js_GC(),* since some of the data it uses has already been de-allocated. Our approach helps to expose the thread interleaving leading to the crash. However in this bug, our approach does not expose the exact location of the crash, *js_SweepScriptFilenames()* underlined in the figure, due to the call-depth limit that we have imposed. If we wanted to capture this additional level of detail by increasing the call-depth, then the trace would become too large (4800 call + return events) and not fit the event queue. The reason is that function *js_GC()* performs a lot of

repetitive work and calls many functions in a loop, which polluted our trace. Fortunately, the exact crash location can be obtained by a regular stack dump, which complements our approach.

	Thread 1: Next-to-last	Thread 2: Last
1		< js_DestroyContext /* last */
2	< js_DestroyContext /* not last*/	
3	. < js_GC	
4		. < ***js_FinishRuntimeNumberState***
5		. > ***js_FinishRuntimeNumberState***
6		. < js_GC
7		. > js_GC
8	// Crash	> js_DestroyContext
9	. . < ***js_SweepScriptFilenames***	

Figure 25. Time-ordered trace for Mozilla bug 515403. Function *js_SweepScriptFilenames* underlined in the trace is not captured by our trace, due to our limit on call-depth.

Another, very interesting order violation bug that we studied is Mozilla bug 388714. This bug was very difficult to debug, taking about a year after it was first reported. Moreover this bug caused a lot of angry Mozilla customers, some even threatening to stop using Mozilla. The reason is that when the bug was triggered, by pressing the refresh button on a web-page, it caused Flash banners to appear blank. This, of course is unacceptable, since advertisers paid for these banners.

In Mozilla, Flash banners live in shells called *iFrame*. An *iFrame* is created by a call to function *doCreateShell()* and then it is populated with content by a call to function *OnStartRequest()*. During a buggy thread interleaving, function *OnStartRequest()* may execute before function *doCreateShell()* and thus attempt to stream content (flash movie) to a frame that doesn't exist.

To debug this problem, we would send a signal to Mozilla to dump a time-ordered trace, once we see the blank banner. Unfortunately, in our experiments we were not able to generate a trace for this bug, since our pin-tool was not able to link the PCs of instructions to the

corresponding source-code functions. We tried compiling Mozilla with a static build, which would help identify the functions corresponding to each PC. However, the Mozilla version containing this bug also contained other bugs which prevented us from generating a static build. However, based on our understanding of the code, and based on discussions and stack dumps provided in the bug report, we believe that our approach would be helpful in this case. In particular, if we dump a trace after a flash banner appears blank, we should be able to observe the flipped order of execution of functions *doCreateShell* and *OnStartRequest* in our time-ordered traces.

4.4.4. Case Study: Logical Concurrency Bug

One class of concurrency bugs that we encountered, and that has received little attention in the research community, is a type of bug that we call *logical concurrency* bug. In this type of bugs, the thread collaboration and interleaving are legal and allowed so that concurrency is promoted. However, certain thread interactions to shared data structures are not well anticipated and may result in incorrect results. To make our discussion concrete, we look at an example.

MySQL server maintains a software structure called Query Cache (QC) in order to improve the performance of some queries. For example, if we issue a *SELECT* query to the server, the server retrieves the data from file, but it also buffers the data in the query cache for possible future reuse. If we issue the same *SELECT* query a second time and the database has not changed in the meantime, the data will be read from memory instead of the file system. However, the query cache has to be used with care in concurrent scenarios, especially if some of the threads are modifying data, which is exactly the issue with MySQL bug 12385. The problem is illustrated in Figure 26, and involves two users submitting commands to MySQL server. First,

user1 obtains a *READ LOCAL* lock on table *t1*. This lock allows other users to perform concurrent inserts on the table. However, these inserts will not be visible to user holding the lock. Next, user2 performs some inserts to table *t1*. Then user1 queries the data in *t1*, and places the results into the query cache. Because of the read-local lock, this data does not contain the inserts from user2. Finally, user2 queries the data and expects to get the latest copy. Instead, user2 receives stale data from the query cache. This bug is fixed by disabling the query cache when the read-local lock is held, and letting the user obtain a fresh copy of the data from file.

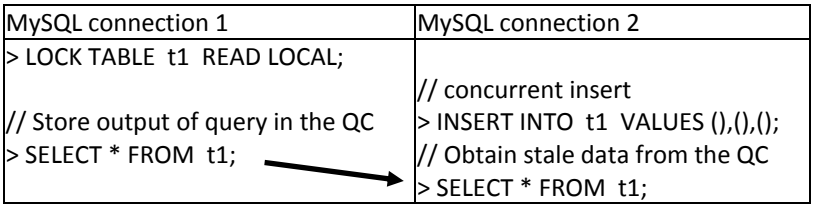

Figure 26. MySQL bug 12385, showing the interleaving of commands issued by two users connected to the server. We assume that a table named *t1* already exists in the database.

Our debugging strategy in this case, is to dump a time-ordered trace immediately after user2 discovers the incorrect results. This allows the user to determine where the results came from and the possible cause of the bug. We achieve that by enabling an always-ON trace and sending a signal to MySQL to dump the trace after we get the incorrect results. Figure 27 shows a simplified version of the trace that we obtained. The last several commands from thread 2 (corresponding to user2) clearly reveal that it has obtained its data from the query cache (functions are marked in italics in the figure). Going back a little further in history and we can find the thread responsible for placing the data into the query cache, command *store_query()* in thread 1. In this particular bug, the thread interleavings are rather coarse-grain, thus even a regular (not time-ordered) trace would probably be sufficient to reveal the issue. In this case our approach is still beneficial, since it enables the *efficient* trace collection.

	Thread 1 SELECT	Thread 2 SELECT (reads QC)
1	< dispatch_command()	
2	. < mysql_parse()	
3	. . < mysql_exec_command()	
4	. . . < open_and_lock_tables()	
5	. . . > open_and_lock_tables()	
6	. . . < select_result()	
7	. . . > select_result()	
8	. . . < ***QC::store_query()***	
9	. . . > ***QC::store_query()***	
10	. . . < handle_select()	
11	. . . < handle_select()	
12	. . > mysql_exec_command()	
13		< dispatch_command()
14		. < mysql_parse()
15		. . < ***QC::send_result_to_client()***
16		. . > ***QC::send_result_to_client()***
17		. > mysql_parse()
18		> dispatch_command()

Figure 27. MySQL bug 12385 showing the threads involved.

Another logical concurrency bug that we investigated is MySQL bug 28249, which is also related to the query cache. This bug is more involved and requires at least three threads to reproduce. It occurs under the following interleaving. First, thread1 issues a *SELECT* statement joining two tables, *t1* and *t2*. The command obtains a lock on table *t1* and opens table *t1* for reading. However, table *t2* is already locked exclusively by thread2, causing thread1 to wait for the lock to be released. In the mean time, thread3 performs concurrent inserts to *t1*. After thread1 obtains the lock on *t2*, it completes the command and places the results in the query cache. The problem is that thread1 opened table *t1* for reading before the concurrent inserts, thus placing stale data in the query cache. To detect this problem, thread1 must check the size of the table (to detect the concurrent inserts) before placing the data in the query cache. To debug this problem, we dumped a trace of the last N event after obtaining the incorrect results. Similarly to the previous example in Figure 27, our approach captures the interleaving of the threads and reveals

that the results are supplied by the query cache. In addition, it reveals that thread3 performed concurrent inserts while thread1 was waiting for a lock, and that thread1 is responsible for placing the data into the query cache.

We believe that capturing such complex logical bugs using a fully automated approach is very difficult, since understanding of the bug requires semantic knowledge of the program. For instance, the query cache was designed to be a concurrent data structure and simply monitoring data races to this data structure is not likely to reveal the bug. Our approach on the other hand is valuable is these cases, since it brings out the non-determinism or thread interleaving and facilitates the programmer in searching for the root cause.

4.4.5. Case Study: Concurrency Bugs Difficult to Debug With Time-Ordered Traces

In our experiments, we found that time-ordered traces are most useful for debugging when the programmer inspects the sequence of the last N events leading up to a program failure, such as assertion failure, or crash. Alternatively, time-ordered traces are useful when the programmer has a specific hypothesis in mind and wants to observe the interleaving of events during a particular execution period, such as a function call. However, just as any other debugging primitive (e.g., watches, stack dump), time-ordered traces may not always be the best or most direct approach to debug a problem. In particular, in some cases the tracing region specified by the programmer may become too large and extremely tedious to examine. In other cases, there simply is a more direct approach to debugging the problem than by using a trace. To illustrate this issue, consider the following example.

In this example, we present a MySQL bug, which may lead to a corrupted database backup. Figure 28 shows a sequence of commands submitted by two users to a MySQL server.

The first user intends to perform a database backup and thus she issues the command *FLUSH TABLES WITH READ LOCK*. This command closes all open tables and acquires a global read lock. A read lock still allows other users to read from the tables, but not modify them. Next, the second user opens a connection to the database and issues a *RENAME TABLE* command. This command is expected to block until the global read lock is released. However, the implementation of the command does not follow the correct protocol and never checks the global read lock. Instead, the rename command succeeds immediately and renames a table potentially in the middle of a backup operation. The simple fix to this bug is to force the rename command to acquire the global read lock before proceeding.

The failure symptom of this bug is a potentially corrupted backup. In this case, if the programmer generated a trace covering the entire backup operation, the trace would be very large and tedious to examine. A much more direct approach to debugging this problem would be to examine the corrupted backup or the binary log and narrow down the problem to the renamed table and thus the RENAME operation. Alternatively, if the developer already knows that the problem lays in the RENAME command, then generating a trace during the execution of that command still does not reveal any additional useful information.

MySQL connection 1	MySQL connection 2
> FLUSH TABLES WITH READ LOCK > perform database backup > UNLOCK TABLES	> RENAME TABLE a TO b;

Figure 28. MySQL bug 2397. RENAME TABLE is not blocked by FLUSH TABLES WITH READ LOCK.

4.5. Limitations and Future Work

As addressed in Section 4.4.5, due to the limited size of our on-chip event queue, the amount of history retained in FIFO mode may not be sufficient to reveal the bug in some cases. In other cases it may become too tedious for the programmer to examine large number of events presented in the trace. We believe that a promising direction to address those issues is to automatically classify which events are likely to be "un-interesting" and remove them from the trace. For instance, we may use the compiler to automatically determine which functions access shared resources. Such functions will be retained in the trace, and the other functions discarded. Such an approach will decrease the clutter of un-important events in the trace, and increase the amount of history that we can buffer. We implemented a simple filter, which removes functions which only access the stack and no other memory. Our preliminary results are promising and show that even such a simple filter can eliminate about 20% to 30% of function calls in some traces. The developer could also specify which images/libraries should be traced, thus eliminate un-interesting events in functions like *printf()* for example. In addition to filtering of un-interesting events, browsing thought large amounts of trace will be significantly facilitated, if the user can zoom-in and zoom-out in terms of detail and call-depth. This can be achieved by an external tool which parses the trace and provides a friendly graphics user interface. As one such prototype, we have developed a tool which parses the trace file into an HTML and Javascript web-page and allows the user to view the trace in a browser and also collapse and expand functions calls/returns and arbitrary code regions in the trace.

Another future direction that we envision for this work is a mechanism to support program evolution and not just debugging. For instance, once it is discovered that a certain

thread interleaving is illegal, we could create a "concurrent assertion", which enforces that rule. For example: *assert(I execute function B, only after function A has already been executed in another thread)* – to verify an interleaving and help with order violation bugs.

4.6. Summary

In this chapter, we present a new debugging primitive for parallel programs. With lightweight architectural support, the proposed primitive can be used to generate a time-ordered event trace to expose thread interleavings of interest. We evaluate the primitive with a variety of concurrent bugs and our results show that the debugging process can be significantly facilitated with time-ordered function call/return traces and time-ordered access traces to specified memory addresses. Overall, based on its effectiveness and the low cost, we make a case for the debugging primitive to be supported in future multi-core processors.

CHAPTER 5. CONCLUSIONS

Software defects and transient faults affecting the microprocessor are major obstacles to system reliability. In this dissertation, we proposed architectural support for approaches to opportunistically detect and correct soft errors. We also proposed architectural support for facilitating or automating the task of software debugging.

The first approach that we proposed, utilizes program localities (or invariants) to detect abnormal execution behavior (anomalies) and to opportunistically detect and correct soft-errors in many processor structures and significantly improve the MTTF (mean time to failure) of the microprocessor. Our experimental results show that the MTTF of the issue queue and the functional units is improved by 39% and 72% respectively. The same hardware support can also be flexibly reused to facilitate automated bug detection, by reporting to the programmer anomalies which occurred during program execution.

The second approach that we proposed, improves on anomaly based software bug detection by aggressively reporting multiple kinds of anomalies and then automatically discarding false-positive anomalies. Our approach consists of three automated steps: bug prediction, isolation and validation. In the first step, we utilize multiple anomaly detectors to monitor for different suspicious behaviors during program execution. In the second step we construct dynamic forward slices from the reported anomalies and we observe which forward slices lead (have a data-dependency relationship) to the program failure. If the forward slice of the anomaly does not lead to the program failure, then the anomaly is discarded as a false-positive. In the third step, we validate the remaining anomalies, by dynamically nullifying the anomalous instruction (turning it into a no-op) and observing if the program still fails. If the

program failure disappears, then we can be confident that we have pin-pointed the root cause of the bug causing the failure. Our experimental results show that our approach is able to accurately pin-point the root-cause of several real-world memory corruption bugs, as well as an infinite loop in the *gcc* compiler.

In our third approach, we address the problem of debugging multi-threaded programs and concurrency bugs. We proposed a new debugging primitive, which allows the programmer to collect time-ordered traces of function calls/returns, memory accesses or possibly other events. Due to its low performance overhead, trace collection can be always-ON even in a production environment, buffering the last N events in each thread of the program. Our experience with concurrency bugs from MySQL and Mozilla show that time-ordered event traces can facilitate debugging in a variety of scenarios, such as: deadlocks, atomicity bugs, order violations and logical concurrency bugs. Some of those bugs cannot be detected by current state of the art automated approaches. Based on the light-weight architectural support, low performance overhead and the utility of this debugging primitive, we make the case to include it in next generation microprocessors.

LIST OF REFERENCES

[1] H. Agrawal and J. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990.

[2] Gautam Altekar, and Ion Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In Proceedings of the *22th ACM Symposium on Operating Systems Principles*, 2009.

[3] T. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd International Symposium on Microarchitecture*, 1999.

[4] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. Unraveling data race detection in the Intel thread checker. In *Proceedings of the 3rd Workshop on Software Tools for MultiCore Systems in conjunction with the IEEE/ACM International Symposium on Code Generation and Optimization*, 2008.

[5] A. Barr. Find the Bug. *Addison-Wesley*, 2004.

[6] Lewis Bil. Debugging Backwards in Time. In *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*, 2003.

[7] R. O'Callahan, J. Choi. Hybrid Dynamic Data Race Detection. *in Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[8] H. Cleve and A. Zeller. Locating Causes of Program Failures. In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[9] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of*

the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2002.

[10] J. Crandall and F. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture*, 2004.

[11] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proceedings of the 34th International Symposium on Computer Architecture*, 2007.

[12] M. Dimitrov and H. Zhou. Locality-based information redundancy for processor reliability. In *Second Workshop on Architectural Reliability in conjunction with the 39th International Symposium on Microarchitecture,* 2006.

[13] Dawson Engler, and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[14] M. Ernst, J. Cockrell and W. Griswold. Dynamically discovering likely program invariants to support program evolution. In *IEEE Transactions on Software Engineering, vol. 27, no. 2, pp. 99-123* , 2001.

[15] M. Ernst, A. Czeisler, W. Griswold and D. Notkin. Quickly Detecting Relevant Program Invariants. In *Proceedings of the International Conference on Software Engineering*, 2000.

[16] C. Flanagan and S. N. Freund. Atomizer: a Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 31st Symposium on Principles of Programming Languages,* 2004.

[17] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating Faulty Code Using Failure-Inducing Chops. In *20th IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[18] M. Gomaa and T. Vijaykumar. Opportunistic Transient-Fault Detection. In *Proceedings of the 32nd International Symposium on Computer Architecture,* 2005.

[19] M. Gomma, C. Scarbrough, T. Vijaykumar and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*, 2003.

[20] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering,* 2002.

[21] M. 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.

[22] D. Hower and M. Hill. Rerun: exploiting episodes for light weight memory race recording. In *Proceedings of the 35th International Symposium on Computer Architecture,* 2008.

[23] D. Jeffrey, N. Gupta and R. Gupta. Identifying the root causes of memory bugs using corrupted memory location suppression. In *Proceedings of the 24th IEEE International Conference on Software Maintenance*, 2008.

[24] B. Korel and J. Laski. Dynamic program slicing. *Journal of Information Processing Letters, Vol. 29, No. 3*, 2008.

[25] T. LeBlanc and J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers, Vol. 36, No. 4*, 1987.

[26] M. Lipasti, C. Wikerson and J. Shen. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems,* 1996.

[27] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools in conjunction with the ACM SIGPLAN Conference on Programming Language Design and Implementation ,* 2005.

[28] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems ,* 2006.

[29] S. Lu, S. Park, E. Seo and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems ,* 2008.

[30] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. Popa, and Y. Zhou. MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles,* 2007.

[31] B. Lucia, J. Devetti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *Proceedings of the 35th International Symposium on Computer Architecture,* 2008.

[32] 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. In *ACM SIGPLAN Conference on Programming Language Design and Implementation,* 2005.

[33] E. Marcus and H. Stern. Blueprints for high availability. *John Willey & Sons*, 2000.

[34] G. Misherghi and Z. Su. Hierarchical delta debugging. In *Proceedings of the 28th International Conference of Software Engineering*, 2006.

[35] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th International Symposium on Computer Architecture*, 2008.

[36] P. Montesinos, M. Hicks, S. King and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *Proceedings of the 14th Architectural Support for Programming Languages and Operating Systems,* 2009.

[37] S. Mukherjee, M. Kontz and S. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[38] Vijay Nagarajan, and Rajiv Gupta. ECMon: Exposing Cache Events for Monitoring. In *Proceedings of the 36th International Symposium on Computer Architecture,* 2009.

[39] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems,* 2006.

[40] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd International Symposium on Computer Architecture,* 2005.

[41] National Institute of Standards and Technology (NIST), Department of Commerce. Software Errors Cost U.S. Economy $59.5 Billion Annually. In *NIST news release 2002-10*, 2002.

[42] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *ACM SIGPLAN Notices, vol 26, no. 7*, 1991.

[43] M. Olszewski, J. Ansel, S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009 .

[44] J. Oplinger and M. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[45] S. Park, and Y. Zhou. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceeding of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[46] E. Pozniansky and A. Schuster. Efficient on-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proceeding of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[47] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *ACM SIGOPS Operating Systems Review, vol. 30, no. SI, pp. 47-57,* 1996.

[48] M. Prvulovic and J. Torrellas. ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30nd International Symposium on Computer Architecture*, 2003.

[49] P. Racunas, K. Constantinides, S. Manne and S. Mukherjee. Perturbation-based fault screening. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, 2007.

[50] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, 2003.

[51] M. Rieker. Advanced Programmable Interrupt Controller. http://osdev.berlios.de/pic.html, 2009.

[52] E. Rotenberg. AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing,* 1999.

[53] H. Saal and I. Gat. A hardware architecture for controlling information flow. In *Proceedings of the 5th International Symposium on Computer Architecture,* 1978.

[54] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. In *ACM Trans. Comput. Syst., Vol. 15, No. 4,* 1997.

[55] Y. Sazeides and J. Smith. The predictability of data values. In *Proceedings of the 30th International Symposium on Microarchitecture,* 1997.

[56] T. Sherwood, E. Perelman, G. Hamerly and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[57] S. Sidiroglou, M. Locasto and A. Keromytis. Hardware support for self-healing software services. In *Workshop on Architectural Support for Security and Anti-Virus in*

conjunction with the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, 2004.

[58] N. Sidwell, V. Prus, P. Alves, S. Loosemore, and J. Blandy. Non-stop Multi-Threaded Debugging in GDB. http://sourceware.org/ml/gdb/2007-11/msg00198.html, 2008.

[59] A. Sodani and G. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th International Symposium on Computer Architecture,* 1997.

[60] D. Sorin, M. Martin, M. Hill and D. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th International Symposium on Computer Architecture,* 2002.

[61] G. E. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems,* 2004.

[62] F. Tip. A survey of program slicing techniques. In *Journal of Programming Languages, Vol.3, No. 3,* 1995.

[63] N. Wang and S. Patel. ReStore: Symptom Based Soft Error Detection in Microprocessors. In *Proceedings of the International Conference on Dependable Systems and Networks,* 2005.

[64] C. Weaver, Joel Emer, S. Mukherjee and S. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31th International Symposium on Computer Architecture*, 2004.

[65] M. Weiser. Program Slicing. In *IEEE Transactions on Software Engineering, Vol. SE-10, No. 4*, 1982.

[66] Y. Xie and D. Engler. Using redundancies to find errors. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002.

[67] M. Xu, R. Bodik, and M. Hill. Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[68] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th International Symposium on Computer Architecture*, 2003.

[69] M. Xu, R. Bodik, and M. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[70] Y. Yu, T. Rodenheffer, W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, 2005.

[71] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002.

[72] A. Zeller. Yesterday my program worked. Today, it does not. Why?. In *Proceedings of the 7th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1999.

[73] A. Zeller. Why programs fail: a guide to systematic debugging. *Morgan Kaufmann*, 2005.

[74] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. In *IEEE Transactions on Software Engineering, Vol. 28, No. 2*, 2002.

[75] X. Zhang, N. Gupta and R. Gupta. Pruning dynamic slices with confidence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[76] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004.

[77] 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. In *Proceedings of the 37th International Symposium on Microarchitecture,* 2006.

[78] P. Zhou, F. Qin, W. Liu, Y. Zhou, J. Torrellas. iWatcher: efficient architectural support for software debugging. In *Proceedings of the 31th International Symposium on Computer Architecture,* 2004.

[79] Developers Guide to WER. https://winqual.microsoft.com/help/default.htm#Developers_Guide_to_WER.htm. 2006.

[80] IA-32 Intel Architecture Software Developer's Manual, Volume 3A: System Programming Guide. Chapter 10, 2009.

[81] Java technology, IBM Style: Monitoring and problem determination. http://www.ibm.com/developerworks/java/library/j-ibmjava5/, 2006.

[82] Polymorph. http://polymorph.sourceforge.net/. 2006.