Unified Architectural Support for Soft-Error Protection or Software Bug Detection

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

Abstract

In this paper we propose a unified architectural support that can be used flexibly for either soft-error protection or software bug detection. Our approach is based on dynamically detecting and enforcing instruction-level invariants. A hardware table is designed to keep track of run-time invariant information. During program execution, instructions access this table and compare their produced results against the stored invariants. Any violation of the predicted invariant suggests a potential abnormal behavior, which could be a result of a soft error or a latent software bug.

In case of a soft error, monitoring invariant violations provides opportunistic soft-error protection to multiple structures in processor pipelines. Our experimental results show that invariant violations detect soft errors promptly and as a result, simple pipeline squashing is able to fix most of the detected soft errors. Meanwhile, the same approach can be easily adapted for software bug detection. The proposed architectural support eliminates the substantial performance overhead associated with software-based bug-detection approaches and enables continuous monitoring of production code.

1. Introduction

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 [10] and incur high cost, an estimate of $59.5B a year, on US economy [12]. 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 [19],[20], which further exacerbates the problem of system reliability.

While software bugs and soft-errors have unrelated origins, they share common traits of manifestation. 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 paper, 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 detection: if an instruction satisfies a certain pattern or a locality, any diversion from this pattern could indicate abnormal behavior. In this paper 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 encoded one, we can speculate that an exceptional event has occurred. The cause of the exceptional event 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. 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 violations are logged to facilitate software debugging.

Compared to traditional soft-error protection approaches, which utilize space or time redundancy [1][7][11][15], we consider our approach an information redundancy scheme, which encodes the proper instruction

1089-795X/07 $25.00 © 2007 IEEE
DOI 10.1109/PACT.2007.15

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 [5]. 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) [22], to provide developers with additional information about program behavior or possible causes of an application failure.

The rest of the paper is organized as follows. Section 2 defines the LVDV locality. Section 3 discusses the related work on soft-error detection and software bug detection. Section 4 describes our proposed architectural support. Section 5 shows how our design protects processor pipelines from soft-errors and Section 6 shows how it is possible to track down software bugs with the same architectural support. Section 7 concludes the paper.

2. 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 *XOR*ing the two values. LVDV extends the traditional/classical value localities [8][16] 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, although there seems to be no apparent value pattern, an output of 100000000014 still hints 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 can not 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.

3. Related work

3.1. Locality-based soft-error detection

Implicit redundancy through reuse (IRTR) [6] utilizes instruction reuse [18] 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 [19]. 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.

Compared to our preliminary study on utilizing LVDV locality for soft-error protection [2], this paper refines the experimental methodology to perform more accurate fault

injection and provides a more complete set of experimental results. Concurrently to our study, a similar idea was independently proposed by Racunas el at. [14]. 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 paper, 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.

3.2. Locality-based software bug detection

Program localities, invariants in particular, have previously been exploited by software-based approaches such as DAIKON [3][4] and DIDUCE [5] to discover software bugs. It has been shown that invariant violations are especially helpful to pinpoint latent code errors [5]. 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. It requires no program instrumentation/recompilation, thereby being 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. [13] 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 [13] 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 [21]. 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.

4. 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 to 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, a violation 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.

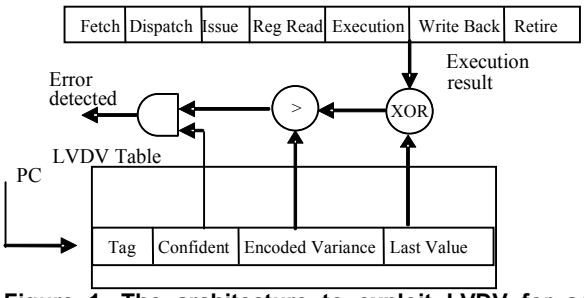

Figure 1. The architecture to exploit LVDV for soft error detection or software bug detection.

5. Soft-error protection

5.1. 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 [19]. 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 paper, we adopt pipeline squashing for its simplicity and our experimental results show that pipeline squashing is capable of fixing many errors, which 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 error 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 5.4), the protection of the issue queue and functional units are examined in detail.

Soft errors in the LVDV table itself are not critical for correctness and can only cause a false-positive violation, or loss of error protection. The LVDV is also not on the critical path of the processor, because it only needs the PC 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. A more detailed discussion on the reliability and complexity impact of the LVDV table is presented in [2].

5.2. 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. 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 [19], 10000 per run is a large enough number of injections to make our results statistically significant. Similar to [19] 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 [2], we perform a larger number of injection runs in this paper. 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.

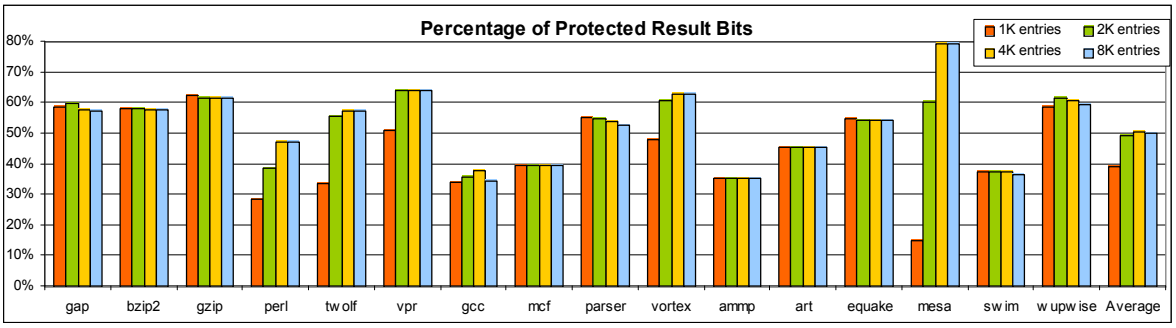

Figure 2. The fraction of protected bits using LVDV locality

5.3. 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 [17] 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).

5.4. Experimental results

5.4.1 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. We also experimented with different chunk sizes 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

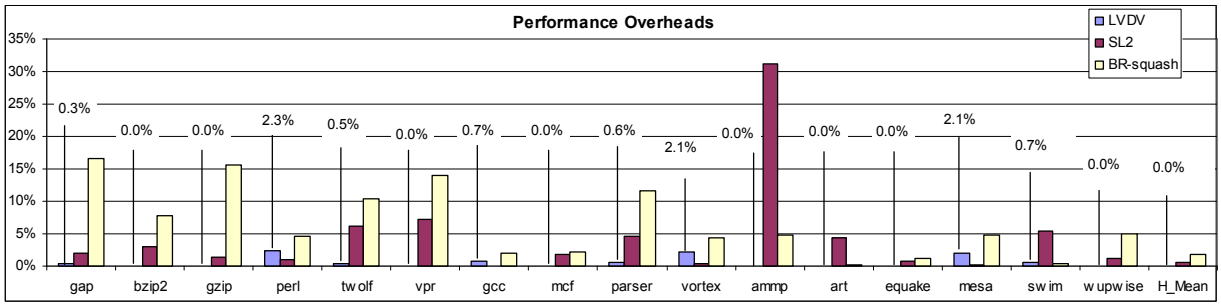

Figure 3. Performance overheads of different error protection schemes

exponent bits on average for the floating point benchmarks, and up to 10 out of 11 for *equake* and *mesa*.

5.4.2 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) [20] and Branch-miss Squash (BR-squash) [19]. The Instruction Redundancy through Reuse (IRTR) [6] approach, detailed in Section 3.1, 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 rational 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 [2], 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 [19]. 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 by 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%.

5.4.3 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

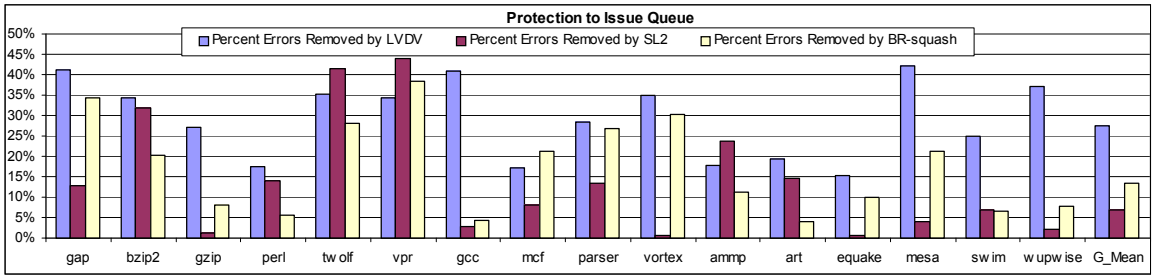

Figure 4. Protection to IQ by LVDV, SL2 and BR-squash

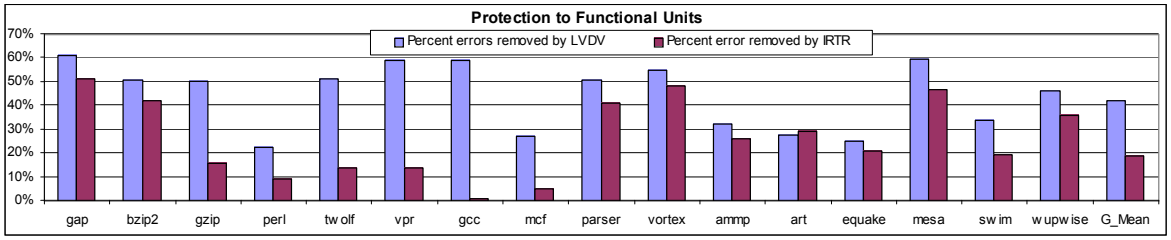

Figure 5. Protection to FUs by LVDV and IRTR

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.

5.4.4 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 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.

6. 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.

6.1. Software bug detection mechanism

As addressed in Section 3.2, 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 [3][4][5][21] 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 violations in subsequent runs. The invariants can also be used to detect violations 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 [21].

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 will be output to a log file. The log for each violation 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 violations can be easily detected and removed by a simple post processing of the bug report. 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 [9]. 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. [5].

6.2. Experimental methodology

To evaluate the effectiveness of our approach, we use four applications from the BugBench benchmark suite [9], *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).

6.3. 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*.

6.3.1 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. The specially crafted input program was largely different from our training set and thus the LVDV tables signaled a large number of violations: 45 and 54 for DIDUCE and for the 4K LVDV respectively (after eliminating duplicate violations with the same PC, and violations 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 variance violations was large, those violations were clustered in several specific functions. Some of those violations were *new-code* violations, which indicated that these regions of code were rarely exercised. As noted by Hangal et al. [5], 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 a violation. In fact, two variance violations 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" [23]. 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 variance violations. The first violation corresponds to the store byte instruction, which stores a character from array *original[i]* to array *newname[i]*. The second violation 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 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

6.3.2 Misuse of library functions, *sprintf* 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 violations 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 violations to function calls *strlen*( ) which computed the length of the input filename elsewhere in the code. Such violations 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*( ) violations. 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 violations. 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 violations 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, the compiler is 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 violations 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 violations only in the application *bc*, compared to DIDUCE. For the rest of the applications, the number of signaled violations was identical as shown in Table 2. The total number of signaled violations is shown after eliminating duplicate violations from the same instruction and violations from external libraries. Since *ncompress* requires different inputs to trigger the bugs, we provide the number of violations signaled for each input. For the rest of the benchmarks, a single input was sufficient to trigger all bugs.

Table 2. Total number of variance violations 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

7. Conclusions

In this paper 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 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.

8. Acknowledgements

We thank Shan Lu from UIUC for providing us with source code, scripts and inputs of the BugBench applications. We also thank the anonymous reviewers for their helpful suggestions on improving our paper.

9. References

[1] T. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design", *MICRO-32*, 1999

[2] M. Dimitrov and H. Zhou, "Locality-based information redundancy for processor reliability", *WAR-2 workshop in conjunction with MICRO-39*, 2006

[3] M. Ernst, et. al., "Dynamically discovering likely program invariants to support program evolution", *IEEE TSE*, Vol.27, No. 2, February 2001.

[4] M. Ernst, et. al., "Quickly detecting relevant program invariants", *ICSE*, 2000

[5] S. Hangal and M. Lam, "Tracking down software bugs using automatic anomaly detection", *ICSE*, 2002

[6] M. Gomaa and T. Vijaykumar, "Opportunistic Transient-Fault Detection", *ISCA-32*, 2005.

[7] M. Gomma et. al., "Transient-fault recovery for chip multiprocessors", *ISCA-30*, 2003.

[8] M.H. Lipasti, C. B. Wikerson and J. P. Shen, "Value locality and load value prediction," *ASPLOS-7*, 1996.

[9] S. Lu, et. al., "Bugbench: Benchmarks for evaluating bug detection tools", *Workshop on the Evaluation of Software Defect Detection Tools,* 2005.

[10] E. Marcus and H. Stern, "Blueprints for high availability", John Willey and Sons, 2000.

[11] S. Mukherjee, et. al., "Detailed design and evaluation of redundant multithreading alternatives", *ISCA-29*, 2002.

[12] National Institute of Standards and Technology (NIST), Department of Commerce, "Software errors cost U.S. economy $59.5 billion annually", *NIST* news release 2002-10, 2002.

[13] J. Oplinger and M. Lam "Enhancing software reliability with speculative threads", *ASPLOS-10*, 2002.

[14] P. Racunas, K. Constantinides, S. Manne and S. Mukherjee, "Perturbation-based fault screening", *HPCA-13*, 2007.

[15] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors", *FTCS-29*, 1999.

[16] Y. Sazeides and J. E. Smith, "The predictability of data values," *MICRO-30*, 1997.

[17] T. Sherwood, et. al., "Automatically characterizing large scale program behavior", *ASPLOS-X*, 2002

[18] A. Sodani and G. Sohi, "Dynamic instruction reuse", *ISCA-24*, 1997.

[19] N. Wang and S. Patel, "ReStore: Symptom Based Soft Error Detection in Microprocessors", *DSN*, 2005.

[20] C. Weaver, et. al., "Techniques to reduce the soft error rate of a high-performance microprocessor", *ISCA-31*, 2004.

[21] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff and J. Torrellas, "AccMon: Automatically detecting memory-related bugs via program counter-based invariants", *MICRO-37*, 2004

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

[23] Polymorph,http://polymorph.sourceforge.net/,2006