Locality-Based Information Redundancy for Processor Reliability

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

ABSTRACT

In this work, we propose a novel information redundancy scheme to protect microprocessors from transient faults. Similar to traditional information redundancy techniques such as ECC, our approach does not require redundant execution in order to detect faults. Instead, redundant bits are used to encode correct instruction execution. However, our mechanism is fundamentally different from traditional information redundancy approaches in how the redundant bits are generated. Rather than using generic information theory, our scheme exploits a program locality, named limited variance in data values (LVDV) to encode instruction execution. In our proposed scheme, a single table tracks the expected behavior of instructions and is used to protect multiple *logic units in the pipeline at the same time. The experimental results show that our proposed scheme significantly improves the MTTF (mean time to failure) of both the issue queue and functional units and provides an efficient mechanism for opportunistic transient-fault recovery.*

I. INTRODUCTION

Under the current trends of technology scaling, transient faults (also known as soft-errors) are predicted to grow at least in proportion to the number of devices [13],[14]. To meet reliability targets, designers need to consider error protection early in the design process at the architecture level, identify critical components, and employ techniques, which offer the right balance between error protection and performance/power overhead. To detect or recover from transient faults, computer systems are typically equipped with a combination of space redundancy [1], [5], time redundancy [8], [9], and information redundancy schemes. Space or time redundancy schemes involve redundant execution to protect processor logic and may incur significant energy or performance overheads. In comparison, information redundancy approaches such as parity or error correction coding (ECC) do not require redundant execution. Instead, they generate redundant bits using the algorithms derived from information theory to detect/recover faults on stored or transmitted data. However, it is commonly regarded difficult for information redundancy schemes to protect computational or control logic functions [6].

In this paper, we advocate the use of locality-based information redundancy approaches to protect processor logic, both computational and control logic. Historically, program localities have been studied extensively and widely used in high performance processor design. We observe that program localities also enable efficient encoding to protect processor logic. In this work, we focus on a novel locality, called limited variance in data values (LVDV), and design a simple yet effective information redundancy scheme for opportunistic transient-fault recovery.

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 *XOR*ing 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, an error is detected speculatively. Then, the pipeline is squashed and the offending instructions as well as other squashed instructions are re-executed in an attempt to correct the error transparently. Our design requires only modest hardware resources and opportunistically protects multiple processor structures including: decode logic, rename tables, the register file, issue queue and functional units.

Program localities, invariants in particular, are also exploited to discover software bugs [2],[3]. The principle of such approaches is to use anomalies in program execution, e.g., a deviation from invariant computation results, as symptoms of potential software bugs. It has been shown that invariant violations are especially helpful to pinpoint latent code errors promptly [3]. The LVDV approach proposed in this paper detects instruction-level invariants to protect hardware logic. Given its similarity to software bug detection schemes such as DIDUCE [3], the proposed scheme can also be used as an efficient hardware accelerator to reduce the performance overhead of software bug detection.

II. Exploiting Value Locality for Reliability

A. Limited Variance in Data Values

The program characteristics termed value locality has been studied extensively [7], [10]. It generally refers to the likelihood that the instructions' execution results exhibit predictable patterns, such as constants, strides, etc. In this work, we realize that value localities enable efficient encoding of instruction execution. For example, if an instruction produces a history of a constant value, the constant value can be viewed as an encoding of the execution process of that particular instruction. Whenever the instruction produces a result that is different from the constant value, it would indicate abnormal behavior or a possible soft error. The problem with traditional value locality is that some instructions do not exhibit strong patterns. In this case, it offers either no protection or generates too many false alarms, which may result in excessively high performance overhead.

In this paper, we propose a new type of value locality named limited variance in data values (LVDV) for soft error protection. Variance between two values is simply defined as the result of *XOR*ing the two values. LVDV extends the traditional/classical value localities and can be exploited for higher protection coverage and lower false positive rates. 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 a soft error. LVDV also captures the region locality, which refers to the fact that memory operations tend to access data in a fixed region. For example, a load instruction 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 error protection. 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 error. 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 paths, which produce the upper 25 bits of the results, are 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. the value changes from 100 to 1) correctly. Since soft errors happen rather infrequently, LVDV presents a more desirable tradeoff between protection coverage and performance overheads.

LVDV can be encoded using the following simple technique. 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* bits instead of 32 bits. The decode process is also 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.

Given the similarity between value locality and instruction reuse [12], it is worth to address the difference between our approach and implicit redundancy through reuse (IRTR) [1]. IRTR stores both operation inputs and outputs in a reuse buffer (RB). In case 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 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 what the expected variance should be. The storage requirement is also reduced compared to IRTR, since we do not need to keep input values.

Exploiting value locality for soft error detection is similar to symptom-based soft error detection, in which mispredictions of high confidence branches are used as symptoms of soft errors [13]. 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 indicated in our experimental results (see Section 4).

Using a ‘signature’ to protect pipeline control logic is proposed in [6]. In this scheme, the control logic signals of an instruction are categorized as either static or dynamic. The static signals used in each pipeline stage are integrated as an instruction’s signature. When an instruction is ready to commit, the history of its static control signals are compared with the cached signature to detect errors. A signature of dynamic control signals, however, can keep changing. As a result, component replication (or space redundancy) is used in [6] to protect dynamic control signals. Compared to this approach, our scheme uses the similar principle of signature checking to detect soft errors but our signature of instruction execution is generated through program locality and is able to protect both static and dynamic signals in control and computational logic.

As addressed in Section 1, the proposed LVDV approach is closely related to software approaches for dynamically discovering program invariants [2],[3]. 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. Once the invariance information is obtained, it can be used as a helpful guide for modifying/evolving the program [2] or detecting latent software bugs [3].

B. *Information Redundancy through LVDV*

Our proposed design to exploit LVDV for soft error detection/recovery is shown in Figure 1. The key component is the LVDV table, which is a cache structure keeping track of variances of various static instructions. Each data entry in the table contains an encoded variance, a last-value field, and a 3-bit saturating confidence counter.

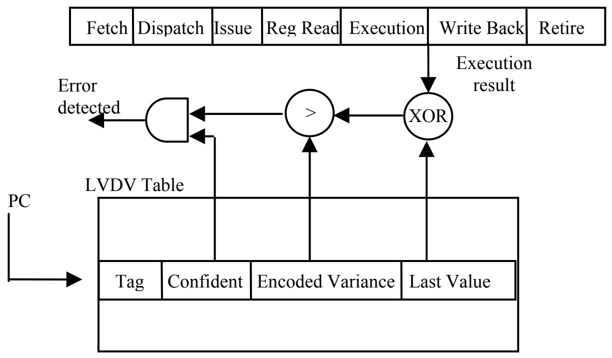

Figure 1. The architecture to exploit LVDV for soft error detection.

Instructions access the LVDV table with their program counter (PC). The variance between the instruction’s last two results is obtained by *XOR*ing 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 equal to 7 (maximum confidence), a possible soft error is detected. If the current variance is larger than the encoded one and the confidence is low, it 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 field is replaced with the current execution result.

When a likely soft error is detected by the LVDV table, the processor can fall back to a previous checkpoint as proposed in [13]. Alternatively, it may squash the pipeline and resume execution from the instruction that resides at the head of the re-order buffer (ROB). At the same time, the new variance is updated to include the faulting chunk and the confidence is reset to zero. 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 and functional units. The reason is that an error is promptly detected if the faulting instruction or one of its immediate dependent instructions has limited variance. Pipeline squashing is usually sufficient to prevent the error from being committed to the architectural state and the re-execution of the faulting instruction will ensure correctness. In the case when the detected error is a false positive, pipeline squashing incurs performance overheads but does not affect correct program execution.

The LVDV table captures the runtime execution behavior of value-producing instructions. 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 Section V, the protection of the issue queue and functional units are examined in detail.

C. *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 error alarm, 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 error, 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 error 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.

III. 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 2000 benchmarks with the reference inputs. Representative simulation points are determined using the SimPoint [11] 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.
Br 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
Mem. Disambig.	Perfect memory disambiguation

The LVDV table that we use has 1024 entries and is configured as 4-way set-associative. Each entry in its data store takes 39 bits, including a 3-bit confidence counter, a 4-bit variance value (i.e., we use 4 chunks to encode the 32-bit variance), and a 32-bit field for the last value. Therefore, the size of the LVDV table is 39k bits. 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, immediate dependent operations offer indirect protection if they exhibit limited variances.

IV. 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 each of the above structures is evaluated separately by injecting at least 10000 errors into the structure under study. According to the analysis in [13], 10000 is a large enough number of injections to make our results statistically significant. Similar to [13] 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 disable the assertions in the timing simulator and let the error propagate. 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 the instruction's operands and opcode. We assume 8-bit operands and 16-bit 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 choose to simplify our implementation and ignore such errors. Due to lack of the circuit implementation details in our timing simulator, we cannot properly model error propagation within combinational logic networks. 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

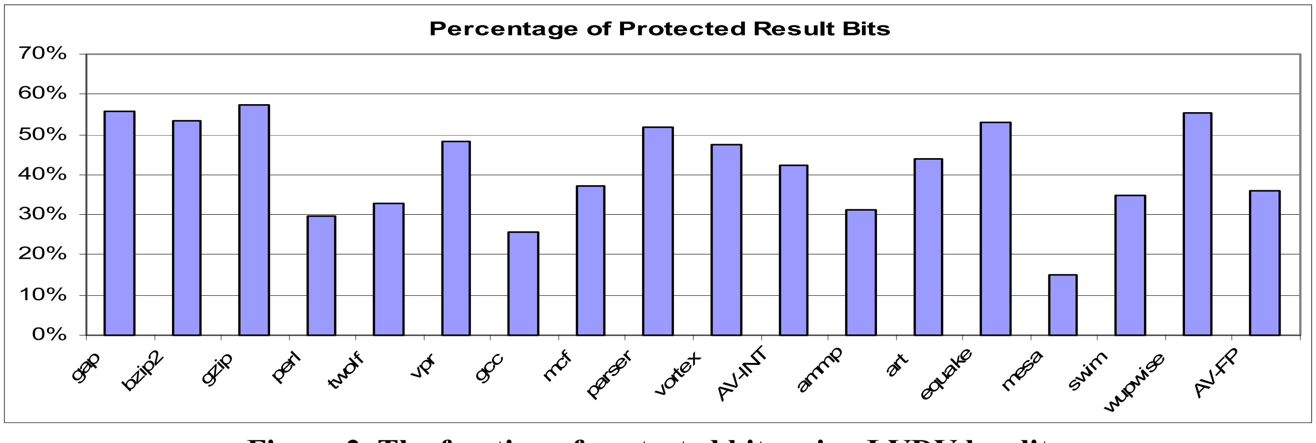

Figure 2. The fraction of protected bits using LVDV locality.

record the number critical faults again. The difference in the number of critical faults shows the effectiveness of the fault-protection scheme. Faults that do not corrupt the architectural state in the base case are not considered.

V. Experimental Results

We first examine 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. The ratio of all the protected bits over the overall result bits is reported for each benchmark, as shown in Figure 2. It can be seen that the proposed LVDV protects a significant portion of execution results, implying strong LVDV locality for various benchmarks.

Next, we examine the protection provided by the LVDV mechanism. In particular, we look at the protection provided to the issue queue (IQ) and the functional units (FUs). We also compare our approach to three other existing approaches: Instruction Redundancy through Reuse (IRTR), Squash on L2-miss (SL2) [14] and Branch-miss Squash (BR-squash). IRTR was detailed in Section 2.A. We implement IRTR as a 1024 entry, 2-way table. Each entry contains two inputs and one output, for a total of 96k bits. 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 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. BR-squash is a modified version of the symptom based protection mechanism proposed in [13]. 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.

We first evaluate the performance overheads introduced by all these protection mechanisms in fault-free environment. In SL2, instruction execution can be significantly delayed since squashing on a L2 miss prevents the processor from performing any overlapping computation. For the benchmark *mcf*, many parallel cache misses become sequential due to the squashing, which results in relatively large performance degradation. In both the LVDV scheme and BR-squash, a pipeline squashing could be due to a false alarm as both schemes are speculative. IRTR, on the other hand, is the only one that does not incur performance overhead. The performance results are show in Figure 3. As it can be seen from Figure 3, the proposed LVDV scheme incurs very limited performance overhead, up to 3.3% in the benchmark *twolf* and an average of 0.7% and 0.3% for the integer and floating-point benchmarks, respectively. Here, the average performance is computed by the harmonic mean of the IPCs and then normalized to the baseline processor (labeled either 'HM_INT' or 'HM_FP').

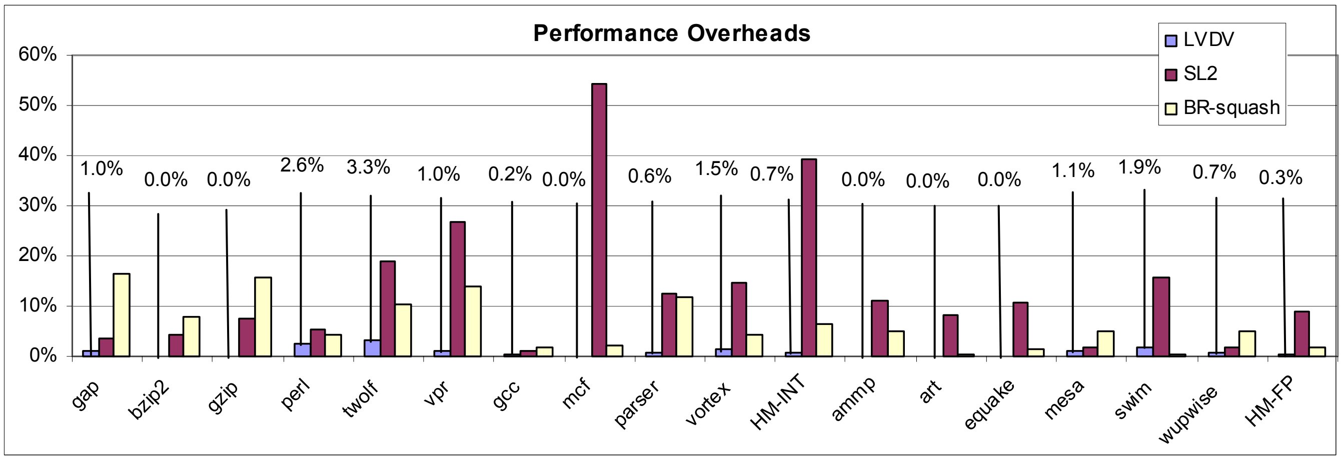

Figure 3. Performance overheads of different error protection schemes.

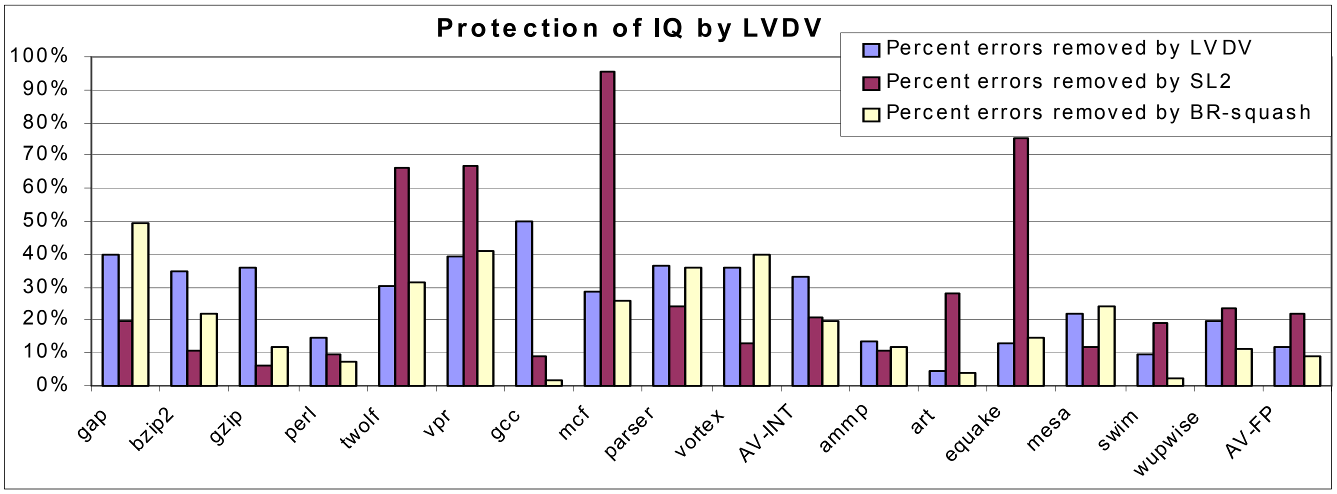

Figure 4. Protection of IQ by LVDV, SL2 and BR-squash.

SL2 incurs very high performance penalties for memory-intensive benchmarks such as *mcf*. On average, 39% and 9% slowdown is incurred for the integer and floating-point benchmarks respectively. BR-squash also has relatively high performance overheads, up to 17% and an average of 7% for the integer benchmarks. The floating-point benchmarks have low branch misprediction rates. Therefore, the overhead is much lower, about 2% on average. Comparing the average false alarm rates, for the integer benchmarks, LVDV generates an average of 0.32 false alarms per 1000 retired instructions while BR-squash has 5.63 false alarms per 1000 instruction and SL2 squashes the pipeline 8.57 times every 1000 instruction. The different false positive rates explain the performance overheads induced by these schemes.

The protection provided to the IQ is detailed in Figure 4. The proposed LVDV approach performs best for the integer benchmarks, removing 33% of critical errors, compared to 21% and 20% for SL2 and BR-squash respectively. LVDV removes 12% of critical errors for the floating-point benchmarks, where SL2 and BR-squash remove 23% and 9% respectively. Notice that SL2 is very effective in protecting those benchmarks, which suffer from a large number of cache misses, such as *twolf*, *vpr*, *mcf* and *equake*. Similarly, BR-squash is effective on benchmarks with many branch mispredictions, such as *gap, parser*, *twolf*, *vpr*, and *vortex*. However both SL2 and BR-Squash suffer from severe performance degradation on these benchmarks, due to the large number of squashes, as reported in Figure 3. Note that, 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 much more promptly and simple pipeline squash can fix them in time, instead of relying on a

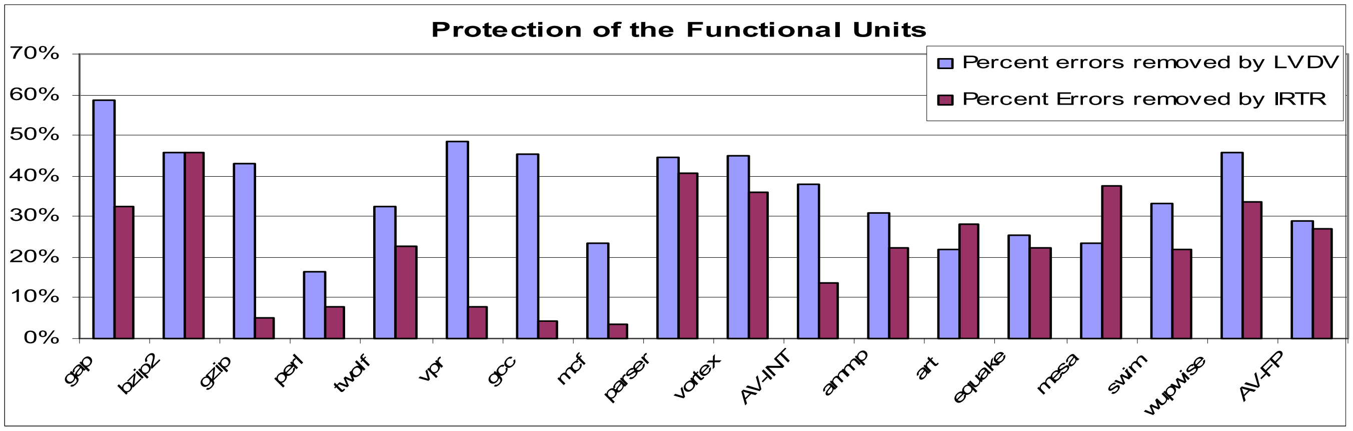

Figure 5. Protection of FUs by LVDV and IRTR.

checkpoint mechanism. By removing 33% of critical errors on average for integer benchmarks, LVDV improves the MTTF (Mean Time to Failure) of the issue queue by 49%. Since MTTF is defined as the inverse of the error rate, the improvement of MTTF is calculated as 1 / (1- %errors removed). Compared to SL2 and BR-squash, our approach is more effective and also provides strong protections to other critical logic units such as functional units as we will examine next.

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 (i.e., the renamed register numbers), IRTR only protects the opcode. In a microarchitecture that models the issue logic using reservation stations, IRTR will be more effective.

In our next experiment, we evaluate the effectiveness of LVDV on FUs as compared to IRTR. We do not include SL2 and BR-squash as these mechanisms did not protect well from the faults injected into the FUs. Figure 5 shows that the proposed LVDV removes many more critical errors than IRTR. It achieves a reduction of critical errors of up to 59% for gap and 38% on average for the integer and 29% for the floating point benchmarks. Considering the MTTF of the FUs, our opportunistic error protection provides up to 144% improvement of MTTF for gap, and 61% improvement of MTTF on average for the integer benchmarks and 40% for the floating point benchmarks.

In general LVDV is more effective on the integer than the floating point benchmarks. The reason lies in the way floating point numbers are represented. Our simple encoding mechanism frequently detects variance in the highest chunk of a floating point value and essentially offers no/little protection. In this work, we have purposely kept our LVDV implementation simple. However we are confident that more intelligent encoding techniques are possible. One possibility is a special variance encoding for floating-point values by separately encoding the variances in their fraction and exponent fields. Such exploration is left as future work.

VI. Conclusion and Future work

We advocate locality-based information redundancy for protecting instruction execution. A new locality, called limited variance in data values (LVDV), is proposed and exploited for opportunistic transient-fault recovery. In our proposed scheme, a single hardware structure which tracks the variance of execution results can protect multiple logic units in the fetch, decode, issue, and execution stages. The experimental results show that the proposed scheme improves the MTTF of both the issue queue and the functional units, by an average of 41% and 61% respectively, and such reliability enhancements are achieved at negligible performance overheads.

The future work is focused on exploring other program localities that can be used for information redundancy. One example is the locality of memory aliasing. If a store and a load always access the same address (e.g., due to spilling and refilling), any exception will indicate a potential error in program execution.

VII. Acknowledgement

We would like to thank the anonymous reviewers for their insightful feedback on our work.

References

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

[2] M. Ernst, J. Cockrell, W. Griswold and D. Notkin, "Dynamically discovering likely program invariants to support program evolution", IEEE TSE, Vol.27, No. 2, February 2001

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

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

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

[6] S. Kim and A. Somani, "On-line integrity monitoring of microprocessor control logic", *ICCD*, 2001.

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

[8] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives", ISCA-29, 2002.

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

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

[11] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior", *ASPLOS-X*, 2002

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

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

[14] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor", *ISCA-31*, 2004.