Quantized AVF: A Means of Capturing Vulnerability Variations over Small Windows of Time

Arijit Biswas, Niranjan Soundararajan, Shubhendu S. Mukherjee, Sudhanva Gurumurthi

***Abstract*— Architectural vulnerability factor (AVF) is the probability that a transient fault in a bit, gate, or transistor becomes a user-visible error. AVFs vary widely across time, applications, and bits. Usually AVFs averaged over time and across applications are used to compute the overall soft error rate of a processor. Average AVFs, however, cannot express the short-term vulnerability variations of a bit as they tend to settle down to a fixed value over time.**

To quantify the vulnerability of bits over short durations, we introduce the concept of Quantized AVF (Q-AVF). Q-AVF expresses the vulnerability of a bit to soft errors over short intervals of time. The average AVF of a bit for a specific interval can be computed as a weighted average of Q-AVFs of all the quanta in that interval. Our analysis of Q-AVF shows significant run-time variation—as much as 80% or more for certain applications. By capturing vulnerability variations over short windows of time, Q-AVFs provide better opportunities for reducing the performance and power overhead of reliability solutions at run-time.

To compute Q-AVF in hardware, linear regression analysis is used to create highly accurate equations that can be implemented with eight simple parameters. These parameters can accurately track Q-AVFs of various structures throughout the processor pipeline. Implementing these equations with fewer parameters is critical to reduce the complexity of run-time Q-AVF tracking, thereby making Q-AVF estimation in hardware practical.

***Index Terms*—architectural vulnerability factor, reliability, soft errors**

I. Introduction

Soft errors induced by alpha particles from packaging and atmospheric neutrons continue to be a challenge for microprocessor designers. Protecting against such errors requires using valuable transistors that could have otherwise been used to increase performance or add other features to the processor. Such protection schemes may also burn power unnecessarily because a single microprocessor is unlikely to experience such a soft error during the vast majority of its lifetime. One approach to reducing the run-time overhead in performance and power of architectural protection mechanisms for soft errors is to dynamically disable the protection mechanism. The opportunity to disable protection mechanisms without greatly sacrificing reliability exists in a program's behavior today because a program's vulnerability to soft errors can vary widely over time [9][13]. This would allow us to disable protection when a program's vulnerability to soft errors is low. We could also turn off the protection mechanism in places where the performance penalty or power overhead of turning on the protection mechanism is prohibitively high.

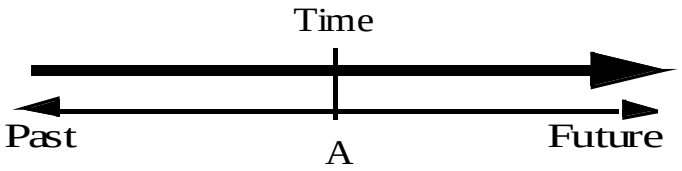

Figure 1. Average AVF Computation requires knowledge of Past & Future

Interestingly, protection mechanisms themselves are often amenable to being turned on and off at run-time. For example, we could dynamically turn off one thread in a Redundantly Multithreaded (RMT) system in which faults are detected by comparing the execution of two identical threads [4]. Similarly, we can turn off error checking code (ECC) activation before a read to save the read latency from a piece of ECC-protected memory, thereby improving performance. We could also decide at run-time whether we want to activate vulnerability reduction mechanisms, such as flushing the pipeline [15]. Many of these techniques have a high cost in terms of power, performance, or both. The ability to disable these mechanisms at run-time allows us to recoup some of this cost of error protection. Previous work has studied these techniques but they have all done so without a means of measuring the impact on run-time reliability. Rather, they focus primarily on optimizing performance. However, without considering the run-time impact to AVF, these techniques run the risk of defeating the very purpose for which the reliability hardware was added in the first place. What is required is the ability to optimize for vulnerability as well as performance

Arijit Biswas is with the SPEARS group at Intel Corporation, Hudson, MA 01749 USA (phone: 978-553-7001; fax: 978-553-2802; e-mail: arijit.biswas@intel.com).

Niranjan Soundararajan is with the Department of Computer Science and Engineering at Penn State University, University Park, PA 16802 USA (e-mail: soundara@cse.psu.edu).

Shubhendu S. Mukherjee is director of the SPEARS group at Intel Corporation, Hudson, MA 01749 USA (e-mail: shubu.mukherjee@intel.com).

Sudhanva Gurumurthi is an assistant professor in the Department of Computer Science at the University of Virginia, Charlottesville, VA, 22904 USA (e-mail: gurumurthi@cs.virginia.edu).

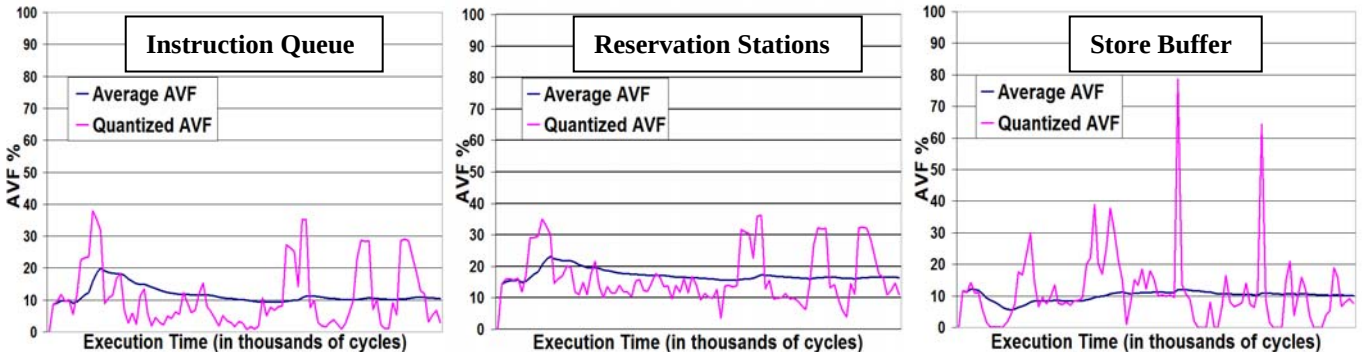

Figure 2. Average AVF & Q-AVF of 3 Structures on a Core™-like processor for a SPEC2K benchmark (parser)

and power. This requires a mechanism to compute the vulnerability variations at run-time, so that we turn off the protection mechanisms without adversely affecting reliability.

This paper focuses on run-time Architectural Vulnerability Factor (AVF) computation, which can be used to decide if an error protection mechanism should be turned off or not at run-time. A transistor device, bit, or gate's soft error rate is computed by multiplying the intrinsic fault rate (e.g., from devices and circuits) and its AVF [7]. AVF is the fraction of faults that result in a user-visible error and is dependent both on a processor's micro-architecture as well as the application running on the processor.

Two key questions underlie run-time AVF computation: (i) what run-time AVF to compute and (ii) how to compute it? These questions are closely related to how we obtain the information to compute the AVF. If we want to compute the run-time AVF at time 'A' in a program flow (Figure 1), then we must know both past and future information.

First, how much of the past do we want to consider to compute the AVF? One approach, such as proposed by Walcott, et al. [13], would be to average the AVF from when a program begins execution, which could mean averaging the AVF over millions to billions of cycles. Unfortunately, such averaging loses the fine-grained variation in AVF. Alternatively, we could compute an instantaneous AVF for each particular cycle. Although instantaneous AVF sounds conceptually appealing, it can introduce too much fluctuation to be of any practical use. Additionally, computing instantaneous AVF would require the run-time AVF mechanism to be triggered every cycle, thereby increasing the mechanism's power overhead.

The second piece of information regarding computing run-time AVFs is knowledge of the future. This is because whether or not a fault becomes a user-visible error depends on the future manifestation of the fault. If a bit in a particular cycle must have the correct value to produce the correct output, then it is required to be correct for Architecturally Correct Execution (ACE). Otherwise, it is un-ACE. How far into the future we have to look to determine a bit's ACE-ness is dependent on the behavior of the bit in the processor. The ACE-ness of a pipeline latch can often be determined fairly quickly, but the ACE-ness of a bit in the data translation buffer may take billions of cycles to resolve [1].

In order to address the issues of needing both past and future knowledge to compute run-time AVFs, we introduce the concept of Quantized AVF (Q-AVF), which averages the AVF over short intervals, such as a thousand cycles. The average AVF over a period of time can be computed by averaging all the Q-AVFs over the same interval. Q-AVF strikes a balance between instantaneous and average AVFs. Because Q-AVF captures the AVF over a short interval, it does not lose the fine-grained variation in AVF (Figure 2). At the same time, Q-AVF does not introduce unnecessary fluctuations that will make it difficult to decide whether to turn a protection mechanism on or off. To the best of our knowledge, no prior work has examined run-time AVFs at such a fine granularity, nor chosen the interval period in such a way as to minimize the effect of both past and future events.

Fu et al. [3] attempted to correlate AVF to a limited set of performance metrics. They chose a small number of metrics, which exhibited inconsistent correlation to AVF across benchmarks, concluded that a single-variable correlation does not exist and did not take a predictive approach. Walcott et al [13] showed that AVF prediction is feasible and the set of metrics required to construct such predictors can be identified automatically through the use of regression analysis techniques. In this paper, we show that using linear regression to compute the average AVF can be extended to Q-AVFs. Specifically, we provide three major contributions beyond Walcott et al's methodology (Co-author Gurumurthi lead the research presented in [13]). First, we study the quantum size over which Q-AVF is computed. The choice of quantum size must expose real-time AVF phase behavior while minimizing noise. This is essential for our goal of using AVF as a practical means of dynamic hardware control. Second, data storage requirements must be finite. Walcott et al carried all past knowledge over each interval while Q-AVF does not. This is important since all intervals are independent, limiting the amount of data that must be retained. Third, we aggregate multiple structures to compute Q-AVF of large portions of a processor, which is important for implementing full-chip protection mechanisms.

Recently, Li, et al. [6] proposed the use of a hardware-based mechanism to compute the average AVFs from a fault-

free execution of the program. Li, et al. attach bits to hardware structures and propagate faults in these bits. From the flow of these bits, one can eventually determine the bit's ACE-ness. In its current form, however, it would be difficult to compute Q-AVFs at run-time using this scheme. This is because we need to inject 100s-1000s of faults before we can compute the AVF of a single structure. By the time we have resolved the ACE-ness of these bits and have computed the AVFs, we would have already executed tens of thousands to millions of cycles, thereby defeating the purpose of Q-AVF to begin with.

This paper makes the following four key contributions:

- We propose the concept of Quantized AVF (Q-AVF) to accurately capture run-time variations in vulnerability. Since it does not average all past history and computes the AVF over small quanta, it succeeds where average AVFs fail. We show why quantum size selection is critical for tracking Q-AVF accurately.
- We demonstrate that Walcott, et al.'s linear regression analysis to compute average run-time AVFs can be extended to compute Q-AVFs. We identify a few simple parameters that can accurately predict the Q-AVF of six key processor structures. Identifying these parameters plays a major role in building a practical hardware-monitor to track Q-AVF variations through the pipeline.
- We show we can aggregate Q-AVFs for regions of a processor, making it easier for a global protection scheme, such as RMT, to decide if protection should be turned on or off for specific regions of an application.
- We provide an example of how Q-AVF control can save power in DMR-based processors. We show that disabling DMR during periods of low Q-AVF can reduce the power consumption of a reservation station by 24%-43%.

The remainder of this paper is organized as follows. Section II is an introduction and evaluation of Q-AVF including analyses of optimal quanta sizes. Section III discusses how to estimate Q-AVF using linear regression analysis to derive equations of hardware-trackable metrics. We provide a detailed analysis of the Q-AVF for six processor structures as well as aggregating these structures to simplify the Q-AVF estimation. We demonstrate a proof-of-concept of how Q-AVF can be used to reduce power in a DMR-based processor. Finally, section IV provides a summary of this work.

II. Evaluation of Quantized AVF (Q-AVF)

Quantized AVF (Q-AVF) is the AVF of a bit or structure computed over defined and moving windows of time called quanta. This section discusses three key aspects of computing Q-AVF: how to extend AVF analysis for Q-AVF, how to select the quantum size for Q-AVF, and how to use linear regression to estimate the Q-AVF at run-time.

A. Extending Average AVF Analysis for Q-AVF

ACE analysis computes the average AVF from a fault-free execution of a hardware model. ACE analysis is usually done on a high-level performance model and tracks the lifetime of a bit in a structure, keeping track of when and what events access that bit. It then determines which components of the lifetime were un-ACE due to such phenomena as killed and dynamically dead instructions. The time the bit spent in ACE state divided by the total lifetime of the bit is the AVF. ACE analysis can be extended using Hamming-Distance Analysis [1] to compute the AVF of address-based structures by identifying only those bits of an address that can cause an actual error.

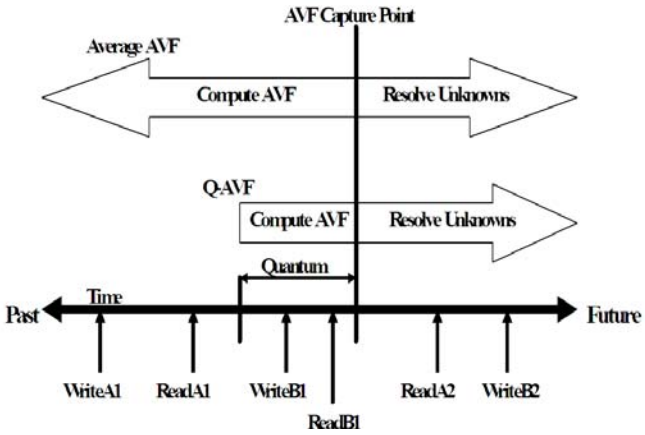

Figure 3. Average AVF & Q-AVF sample timeline. Unknowns resolve to either ACE or un-ACE.

Figure 3 shows the difference between average AVF and Q-AVF computed for the same AVF capture point. Q-AVF only computes the AVF of a structure during one particular quantum of time, ignoring any ACE residence time prior to the start of the quantum. Thus, only ACE residence time that occurs during the quantum affect the Q-AVF computation. In effect, Q-AVF is a piece-wise breakdown of the average AVF. Equation 1 shows how average AVF can be derived from Q-AVF when the quanta sizes are equal.

There is one subtlety in the Q-AVF computation. An ACE entry in a particular quantum could have started its ACE residence time in a previous quantum. Hence, to compute Q-AVF for a given quantum, we assume that any ACE entries resident in the structure from events that occurred prior to the quantum started their residency at the start of the quantum. In Figure 3 the *WriteB1*, *ReadB1*, *ReadA2* and *WriteB2* events will be seen when they actually occurred by both the average AVF and Q-AVF computations. However, the *WriteA1* event will be seen at the correct time by average AVF, but will be seen to occur at the start of the quantum for Q-AVF since the data is resident in the structure during the designated quantum. The *ReadA1* event will only be seen by average AVF and will not be considered by Q-AVF (for the quantum shown in the figure) since it does not impact vulnerability during the

$$Average\_AVF = \frac{\left(\sum_{Quanta} Q - AVF\right)}{Total\_Number\_of\_quanta}$$

Equation 1. Average AVF as a Function of Q-AVF

quantum. Thus, Q-AVF considers only a very short and recent history of the structure's behavior in order to compute the AVF at the same AVF capture point. This assumption still allows Q-AVFs to capture large variations in the structure's vulnerability behavior across consecutive quanta.

Figure 2 shows Q-AVFs for a quantum size of 1024 cycles versus the average AVFs for a SPEC2K benchmark (parser) for three structures. Q-AVF shows clear phases of high and low vulnerability while the average AVF quickly reaches a steady state. This variation suggests that Q-AVF can provide the run-time information on vulnerability changes to allow power, reliability and performance optimizations of error mitigation techniques. For example, this could allow us to enable error protection when Q-AVFs start to rise. Alternatively, this can also allow us to disable the protection mechanism when the Q-AVF falls below a certain threshold, thereby saving the power/performance cost of running the error protection hardware during phases of acceptably low vulnerability. In the next section, we will examine how to compute Q-AVFs.

B. *Selecting the Quantum Size*

To compute AVF—both average and quantized—we require future knowledge to resolve unknown lifetimes of a bit to ACE or un-ACE. Since capturing these future events is difficult, especially in hardware, the choice of quanta size becomes extremely important. A quantum can be as small as a single cycle or as large as the length of the entire application. Both extremes however, introduce problems.

Too small a quanta size will result in Q-AVFs that are dominated by the unknowns since most events that affect the value will occur after the quantum. Soundararajan, et al. [11] showed the vulnerability variations at a single cycle granularity, which was essentially the AVF for a 1 cycle quantum. Soundararajan, et al.'s method suffered from the problem that the unknowns dominate the computation and so the actual vulnerability is difficult to ascertain accurately. Cooldown [1] can reduce the unknowns, but requires knowing the future for millions to billions of cycles, which is not an option for run-time Q-AVF computation.

By contrast, a very large quantum reduces the impact of unknowns significantly since most events that will impact the Q-AVF occur within the quantum and will be accounted for. However, this comes at the cost of losing much of the run-time vulnerability variations thus incurring the same problems average AVF suffers from. Hence, the choice of quanta size is extremely important.

We explored multiple quantum sizes ranging from 64 to 10,000 cycles across all benchmarks in the SPEC2000 and SPEC2006 suites as well as several TPC-C traces. Figure 4 shows the Q-AVF of the micro-op buffer of a Core™-like processor for three different quanta sizes (128, 1024, and 8192 cycles) on one particular TPC-C benchmark to illustrate the selection of an ideal quantum size. The 128 cycle quanta shows significant variability but the amount of unknowns are extremely high. The 1024 cycle quanta has significantly fewer unknowns yet continues to track the Q-AVF variations well.

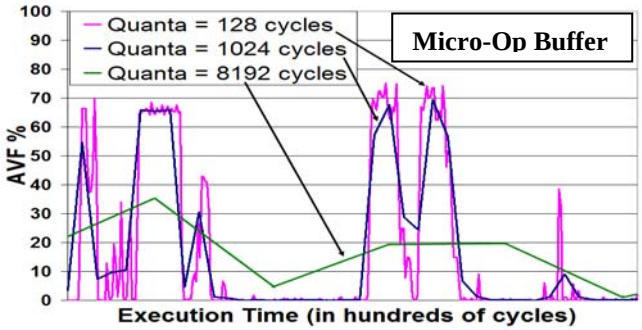

Figure 4. Micro-op Buffer Q-AVFs for different quanta sizes on a Core™-like processor model for a TPC-C trace

The 8192 cycle quanta has almost no unknowns but also no longer shows the different phases of vulnerability. Similar behavior was seen on all benchmarks.

Thus, a large quanta size can significantly reduce unknowns, but comes at the cost of a significant reduction in its ability to track the phases of vulnerability and begins to suffer from the same sort of problems as average AVFs. What is needed is a quanta size large enough to keep the unknowns low, yet small enough that the majority of vulnerability variation can still be observed. The 1024 cycle quanta is ideal for our model and all subsequent data is based on this size.

C. *Reducing Unknowns with Linear Regression*

To use Q-AVF to provide a means of dynamically controlling reliability mechanisms, we must be able to measure the Q-AVF of a given structure or structures at run-time in hardware. Q-AVF, by its very nature, can potentially be tracked in hardware since it is computed over finite, well-defined quanta of time. However the typical means of computing Q-AVF—using ACE analysis or statistical fault injection [14]--are not easily translatable into hardware. Li, et al. [6] describe AVF computation by tracking fault propagation through the hardware using fault-free program execution. But as described in Section I, it would be difficult to use this method to compute Q-AVFs in hardware.

Walcott, et al. [13] showed that linear regression analysis could successfully derive linear equations of processor metrics that are easily measured in hardware. These linear equations were capable of tracking the average AVF values at different points in time with a good degree of accuracy. Essentially, this suggests that Walcott, et al.'s methodology can predict how the unknowns will resolve using current processor metrics. We expanded on this work by using linear regression analysis to derive equations for Q-AVF. Regression analysis allows us to discover the subset of micro-architectural parameters with the strongest correlation to Q-AVF, thus identifying the smallest set of parameters needed to accurately estimate the Q-AVF. We focus on linear regression since this greatly simplifies the hardware implementation of a regression-based Q-AVF estimator, which is one of our end goals.

Walcott et al. simulated portions of benchmarks from the SPEC CPU2000 suite. The AVF for several processor structures was computed every 4 million instructions as an

Instruction Issue	4, out-of-order
I-cache	64 KB, 4-way
D-cache (2 ports)	64 KB, 8-way, 2 cycles
Branch Predictor	Bi-modal (512 entries) + Gshare (1024 entries)
Branch Target Buffer	4 K entries, 16-way
Micro-op Buffer	24 entries
Instruction Queue	14 entries
Reservation Station	32 entries
Retirement Order Buffer	96 entries
Load / Store Buffers	50 / 24 entries
L2 cache	1 MB inclusive, 8-way, 10 cycles
L2 miss latency	200 cycles

Table 1. Parameters for the Core™-like Processor Model

average across this entire interval. The microarchitectural state information was carried over across successive intervals. They then created linear-regression based predictors from this repository of microarchitectural states and AVFs.

We use a similar methodology to compute Q-AVFs for several processor structures. In our case, we simulate portions of benchmarks in the SPEC2000 and SPEC2006 suites as well as several TPC-C traces for varying parameters. We then compute the Q-AVF over 1000 quanta of 1024 cycles each for each benchmark and trace. Micro-architectural state is not carried over from one quantum to the next resulting in each quantum being independent of the one before it. As a result, Q-AVF computation does not average past history and is only a function of the events that occurred during the specific quantum over which Q-AVF is being computed. Descriptions and results of our methodology are given in section III.

We performed this analysis for six structures across an Intel Core™-like processor pipeline. Our resulting Q-AVF estimation showed a high degree of accuracy to the actual Q-AVF, tracking with accuracy better than 0.9 in most cases.

III. Results of Q-AVF Estimation

Section III describes our experimental setup and results. Section III.A describes the processor model and structures studied. Section III.B describes the experimental methodology. Section III.C describes the results of our linear regression experiments on each of the studied structures. We make several observations on the behavior of these structures and how they relate to the Q-AVF and why. Section III.D describes how the previously studied structures can be aggregated in order to measure the Q-AVFs for large portions of the processor pipeline. In this section we present the aggregation methodology as well as the resulting aggregate linear equations, showing that the aggregate equations can be generated using far fewer parameters than the sum total of the parameters in each individual structure's equations while still maintaining high accuracy. Finally, section III.E contains an example of Q-AVF based control of a DMR reservation station resulting in power savings of 24%-43%.

A. Processor Model and Structures

We evaluated Q-AVFs using an Intel Core™-like processor model. The processor model is configured as per the parameters in Table 1. The front end fetches data in-order from the instruction cache and decodes it into x86 macro instructions. These are then placed in the Instruction Queue (IQ) then decoded into micro-ops. The micro-ops move to the Micro-op Buffer (MB) which feeds the back end pipeline.

The back-end executes micro-ops out-of-order. The micro-ops enter in-order, stalling if there is resource contention. The micro-ops then access the Register Alias Table (RAT) to read the location of their source operands. If the source data is ready, the micro-op reads it from the Architected Register File (ARF) or the Physical Register File (PRF)/Retirement Order Buffer (ROB) entries (ROB and PRF are integrated in our model). The ROB maintains the micro-ops in-order for retirement and also holds the result data which is written into the ARF once the micro-op (non-store) commits. The Reservation Station (RS) issues instructions out-of-order to the execution units, once source operands are available.

The memory sub-system provides the actual data required for computation. In our pipeline, the load buffer (LDB) and store buffer (STB) exist as separate structures and make up the memory order buffer (MOB). They act as interfaces to the cache hierarchy and handle memory disambiguation and the ordering of store instructions.

B. Experiment Setup

We performed linear regression analysis using Matlab software on over 150 processor parameters to derive the equations for tracking the Q-AVF in hardware. We simulated Simpoints [10] of 100 benchmarks from the SPEC2K and SPEC2006 suites as well as several TPC-C server traces with varying parameters (such as number of threads and warehouses) on a detailed, industrial-level Core™-like performance model implemented in the ASIM performance modeling framework [2]. The Q-AVF tracking was done over a continuous period of time and not sampled.

To obtain the per-structure parameter correlations with AVF we simulated all traces for 10 million instructions with a further 10 million instructions of cooldown to resolve unknowns. All Q-AVF simulations were done for 1000 consecutive 1024-cycle quanta with an additional 10 million cycles of cooldown to resolve unknowns for each quantum. This is compared against the Q-AVF linear equation results. Since the end goal is to study the feasibility of a hardware Q-AVF estimator, we focus on the estimator's performance on a per-benchmark basis. As such, our results present selected benchmarks that represent the common case correlations as well as the worst cases for each structure. The best case correlations were all above 0.95 with 1.0 being perfect.

C. Per-Structure Linear Regression Analysis

This section summarizes the results of our per-structure correlation analysis. In this paper we study, in detail, the Q-AVF behavior of the Instruction Queue (IQ), Micro-op Buffer (MB), Reservation Station (RS), Retirement-Order Buffer (ROB), Store Buffer (STB) and Load Buffer (LDB). Each structure is broken up into detailed subcomponents whose Q-

Structures	# Params	Linear Equations to Track Q-AVF
IQ	6	A0 - A1 * Total FE Killed Instruction Latency + A2 * IQ Utilization + A3 * Macro Branch Count - A4 * Macro Control-op Count – A5 * FP Micro-op Count - A6 * Store Data Count
MB	5	B0 - B1 * Total FE Killed Instruction Latency + B2 * MB Utilization - B3 * Store Data Count - B4 * Branch Mispredicts + B5 * RS Bypass Count
RS	6	C0 + C1 * RS Utilization + C2 * RS Src Utilization + C3 * Cycles Macro Insts Re-decoded - C4 * Unconditional Branch Predictions + C5 * MB Bypass Cycles + C6 * Decode Stalls
ROB	3	D0 + D1 * ROB Utilization - D2 * Store Data Count - D3 * Immediate-op Count
STB	1	E0 + E1 * STB Utilization
LDB	7	F0 + F1 * Load Replay Count + F2 * L1 Squash Count - F3 * L1 Load Misses + F4 * RS Utilization + F5 * LDB Utilization – F6 * Memory Disambiguation Blocks - F7 * Micro-ops Killed By Br. Mispredicts

Table 2. Linear equations to track the run-time Q-AVF Variations for the different structures
(A#,B#,C#,D#,E#,F# represent the architecture-dependent constants and coefficients)

AVF is computed individually. This allows the Q-AVF computation to be extremely detailed and accurate since even the differing architectural behavior of sub-components is taken into account. For instance, the IQ and the MB are further broken up into immediate fields, control fields, logical source and destination address fields, and valid bits. The RS is broken into sub-fields representing source and destination addresses, control bits, source data for each source, valid bits and flags. The ROB is broken into sub-fields representing allocation control, writeback control, fault and exception fields, data, flags and valid bits. The STB is broken into virtual address, physical address and data fields while the LDB only contains the virtual address. Table 2 gives the linear equations for these structures based on the individual parameter correlations. These correlation values are given in terms of R^2 which is given as a number from 1 to -1 with numbers closer to 0 indicating no correlation and numbers closer to 1 or -1 indicating strong positive or negative correlation. The parameters of the equations given in Table 2 are sorted in order of highest correlation to the Q-AVF of the structure. For example, for the LDB, Load Replay Count shows the highest individual correlation to the LDB Q-AVF followed by L1 Squash Count, followed by L1 Load Misses and so forth.

We list our key observations to provide insight into why the linear regression analysis can estimate Q-AVFs accurately. Our key observations are:

1. AVFs of front-end structures (IQ, MB) correlate highly to structure utilizations further down in the pipeline. Additionally, the RS AVF correlates highly to a front-end parameter (IQ Utilization). This indicates that there is inter-dependence between different events happening at various stages of the pipeline.
2. Multi-correlation runs for the RS gave R^2 values of 0.85 for eight parameters. Sub-component analysis revealed that the source operand array behaved differently from the rest of the RS. Tracking the operands separately improved R^2 values (0.90) and reduced parameters to six.
3. IPC has very low correlation to most structures' AVF since AVF is affected more by other factors like dead code, stalls and other individual structure behavior.
4. Loads in the LDB contribute to AVF only when they miss in the cache or are otherwise blocked. Hence utilization does not have high correlation with the LDB's AVF.
5. The STB's AVF correlates highly to its utilization since STB mostly contains committed stores.
6. The "Total Front-End Instruction Killed Latency" parameter is a composite of 3 parameters: the total instructions in the MB divided by the total decoded macro instructions multiplied by the total flushed instructions. This is approximately the total amount of time that flushed instructions spent in the MB. For two specific SPEC2K benchmarks (vpr and mcf) the addition of this parameter increased the R^2 fit of the linear equations for IQ and MB from 0.6 to better than 0.8.

We find that, while there is significant variability in vulnerability phases across structures and benchmarks, the estimated Q-AVF tracks the actual Q-AVF extremely well. Table 3 shows the mean and minimum correlation values between the actual and estimated Q-AVF on all SPEC2K, SPEC2006 and TPC-C benchmarks for six structures, which span the front-end and back-end, based on 1000 quanta for each benchmark. The table indicates that our Q-AVF estimation tracks the actual Q-AVF with a high degree of accuracy across all simulated benchmarks.

D. Aggregate Structures and Equations

Table 2 shows that, even with the smallest set of parameters, the different structures together require over 20 parameters to be tracked. Expanding the analysis to more structures would introduce more parameters as well. Given that our goal is to build actual hardware, we needed to minimize the set of parameters used to drive the Q-AVF computation. Since the parameter requirements scale with structures, tracking them through the pipeline would increase the hardware and wiring complexity, placing practical

Structures	Mean Correlation across Benchmarks	Min Correlation across Benchmarks
IQ	0.87	0.81
MB	0.90	0.82
RS	0.93	0.82
ROB	0.95	0.86
STB	0.98	0.94
LDB	0.85	0.80

Table 3. Per-Structure Q-AVF Tracking Accuracy

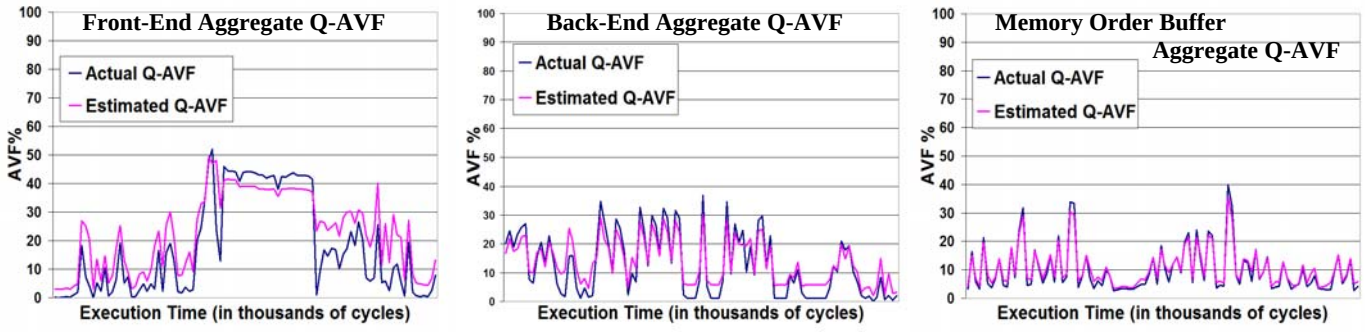

Figure 5. 8 parameter Q-AVF tracking on a Core™-like processor model for a single SPEC2K benchmark (gzip)

limitations on building such a mechanism. While we could drop structures with low average AVFs, these structures could have high Q-AVFs and hence affect actual run-time vulnerability significantly during certain phases of the program. Aggregating the structures provides a better profile of their run-time vulnerability.

Another point to note is that error protection schemes are not always efficient to implement per-structure but makes more sense when used to cover portions of the pipeline such as the front-end or back-end. This does not mean per-structure analysis is not useful. For systems requiring high reliability and those that need to provide very high guarantees (100%) on system vulnerability bounds, the per-structure equations can be built into the hardware mechanism. But for cases where the implementation cost of the hardware mechanism is important, reducing the complexity plays a vital role.

To aggregate the Q-AVFs we combined the Q-AVFs (each structure's Q-AVF is bit-weighted) of MB and IQ into the Front-End (FE) Q-AVF, the RS and ROB into Back-End (BE) Q-AVF and the LDB and STB into the Memory-Order Buffer (MOB) Q-AVF. By aggregating the Q-AVFs of structures with related architectural behavior in the same part of the pipeline, the parameters that track the aggregate Q-AVFs also correlate well to the individual structure's Q-AVFs. These structures cover the major components in each respective aggregate block that contribute significantly to the overall reliability of that block. Reliability schemes such as residue, parity prediction, or redundancy target these structures in some form and can also benefit from dynamic control.

Aggregate Blocks	Linear Equations using the Eight Common Parameters
Front End	A0 - A1 * ST_Flush - A2 * STB_Util - A3 * FE_Kill - A4 * ROB_Empty - A5 * ROB_Util - A6 * Br_Miss + A7 * RS_Util + A8 * IDQ_Util
Back End	B0 - B1 * ST_Flush - B2 * STB_Util - B3 * FE_Kill + B4 * ROB_Empty + B5 * ROB_Util - B6 * Br_Miss + B7 * RS_Util + B8 * IDQ_Util
Memory Order Buffer	C0 - C1 * ST_Flush + C2 * STB_Util - C3 * FE_Kill + C4 * ROB_Empty - C5 * ROB_Util - C6 * Br_Miss - C7 * RS_Util + C8 * IDQ_Util

Table 4. Linear Equations to track run-time Aggregate Q-AVF Variations across the processor pipeline
(A#,B#,C# represent the architecture-dependent constants)

Linear regression analysis on FE, BE and MOB aggregate Q-AVFs identified eight parameters that are common across them. These parameters together track their AVFs with R^2 values better than 0.90. These include:

1. Stores Flushed before DTLB response (ST_Flush)
2. STB Utilization (ST_Util)
3. ROB Empty Cycles (ROB_Empty)
4. ROB Utilization (ROB_Util)
5. Branch Mis-predicts (Br_Miss)
6. RS Utilization (RS_Util)
7. IDQ Utilization (IDQ_Util)
8. Total Front-End Instruction Killed Latency (FE_Kill)

Table 4 shows the linear equations for the FE, BE and MOB using these eight parameters. Figure 5 shows the actual and estimated Q-AVFs for the FE, BE and MOB using the eight parameters on a single benchmark. Table 5 shows the mean and minimum correlations for aggregate Q-AVF tracking across all SPEC and TPC-C benchmarks.

Dynamic Q-AVF control allows us to dial in an acceptable threshold of vulnerability. Should that threshold be exceeded, dynamic Q-AVF control could be used to enable error-mitigation. If the Q-AVF drops below this threshold, error-mitigation could be deactivated, recouping some of the power and performance costs incurred by the mechanism while still guaranteeing a base level of reliability. From Figure 5 if we set the acceptable vulnerability threshold to 10% Q-AVF, we could disable error mitigation 52% of the time in the FE, 43% of the time in the BE and 19% of the time in the MOB.

E. Discussion of Applications of Q-AVF

Dynamic AVF control has significant potential for reducing the power and performance costs of many reliability schemes. Reliability schemes which target large portions of the pipeline

Aggregate Blocks	Mean Correlation across Benchmarks	Min Correlation across Benchmarks
Front End	0.86	0.80
Back End	0.90	0.81
Memory Order Buffer	0.93	0.92

Table 5. Aggregate Q-AVF Tracking Accuracy

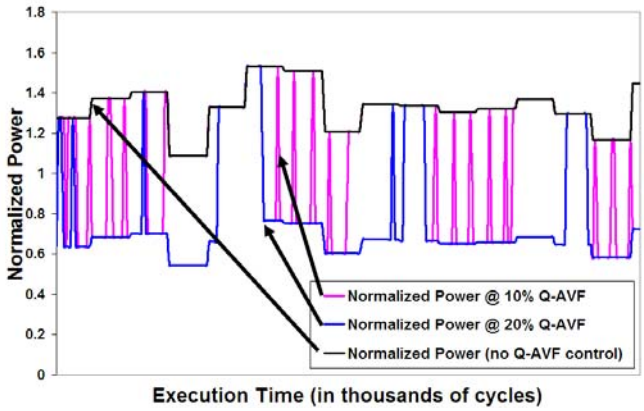

Figure 6. Q-AVF RS Power Reduction of a Core™-like processor model running the swim SPEC2K benchmark

and cover multiple structures [8] could benefit significantly in power savings and performance improvement.

One reliability scheme that demonstrates the advantage of Q-AVFs is microarchitectural redundancy, such as dual-mode redundancy (DMR), which duplicates portions of the pipeline to execute operations redundantly [11]. They are expensive in both power and performance. We present a simple case study as a proof-of-concept that Q-AVF estimation can reduce the power of such schemes while guaranteeing a fixed level of reliability. We implemented a DMR scheme where every RS entry is duplicated on the Core™-like processor model. All accesses to an RS entry also occur in the duplicate entry and data in both RS entries are compared when consumed. As a result, any error in the RS is detected. We use an Intel power model based on ALPS [5] that is closely correlated with silicon. We show the power savings for enable/disable thresholds of 10% and 20%. Figure 6 shows that the dynamic power in the RS halves whenever the Q-AVF drops below threshold, representing intervals when redundancy is disabled. The average power savings is 24% for a Q-AVF threshold of 10%, and 43% for a Q-AVF threshold of 20%. Note that the figure only shows a small portion of the timeline to illustrate the differences in the power while the overall averages are taken over the full 20 million instruction run.

IV. Summary

To optimize reliability techniques used on microprocessors to mitigate the effect of soft errors, we investigated the AVF metric as a potential source of dynamic control. We have shown that average AVF is not an optimal metric to track run-time vulnerability variations due to its tendency to settle at a fixed value over time. We introduced the concept of quantized AVF (Q-AVF), which we showed to better track actual run-time vulnerability variations than average AVF.

We defined the Q-AVF and analyzed the quantum size parameter as well as studied the vulnerability variations of six structures across a Core™-like processor pipeline. To track their Q-AVF in hardware we provided linear equations based on simple pipeline events that could estimate the actual Q-AVF with a high degree of accuracy (within 10%-15% error). We showed that, while per-structure Q-AVFs are interesting, there are practical difficulties in translating them into hardware due to the number of parameters that need to be tracked. Aggregate Q-AVFs overcome this limitation and need only eight parameters to be tracked in order to estimate the Q-AVF throughout the processor pipeline. This reduced the number of Q-AVF estimators from dozens to just three. This is a major step toward implementing a practical hardware Q-AVF estimator in actual systems. Finally, we provided an example which showed that Q-AVF control could reduce the power of a simple DMR system by 24% for a Q-AVF threshold of 10% and 43% for a threshold of 20%.

References

[1] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S.S. Mukherjee, R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures", *International Symposium on Computer Architecture (ISCA)*, 2005

[2] J. Emer, P. Ahuja, N. Binkert, E. Borch, R. Espasa, T. Juan, A. Klauser, C.K. Luk, S. Manne, S.S. Mukherjee, H. Patil, S. Wallace, "Asim: A Performance Model Framework," *IEEE Computer*, 35(2):68-76, Feb. 2002.

[3] X. Fu, J. Poe, T. Li, J. Fortes, "Characterizing Microarchitecture Soft Error Vulnerability Phase Behavior", *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, (MASCOTS)*, September 2006

[4] M.A. Gomaa, T. N. Vijaykumar, "Opportunistic transient-fault detection", *International Symposium on Computer Architecture (ISCA)*, 2005.

[5] S. H. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. "Managing the Impact of Increasing Microprocessor Power Consumption". *Intel Technology Journal Q1 2001*, 5(1), Feb. 2001.

[6] X. Li, S.V. Adve, P. Bose, J.A. Rivers, "Online Estimation of Architectural Vulnerability Factor for Soft Errors", *International Symposium on Computer Architecture (ISCA)*, 2008.

[7] S.S. Mukherjee, C.T. Weaver, J. Emer, S.K. Reinhardt, T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," *International Symposium on Microarchitecture (MICRO)*, December 2003.

[8] A. Parashar, S. Gurumurthi, A. Sivasubramaniam. "SlicK: Slice-based Locality Exploitation for Efficient Redundant Multithreading", *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006

[9] G.A. Reis, J. Chang, R. Cohn, S. Mukherjee, D.I. August, "Configurable Transient Fault Detection via Dynamic Binary Translation", *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[10] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, "Automatically Characterizing Large Scale Program Behavior," *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[11] D.P. Sieworek, R.S. Swarz, "Reliable Computer Systems: Design and Evaluation", A K Peters, *3rd edition*, 1998

[12] N. Soundararajan, A. Parashar, A. Sivasubramaniam, "Mechanisms for bounding vulnerabilities of processor structures", *International Symposium on Computer Architecture (ISCA)*, June 2007

[13] K.R. Walcott, G. Humphreys, S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state", *International Symposium on Computer Architecture (ISCA)*, June 2007

[14] N. Wang, J. Quek, T.M. Rafacz, S. Patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline", *International Conference on Dependable Systems and Networks (DSN)*, June 2004

[15] C. Weaver, J. Emer, S. Mukherjee, S. Reinhardt, "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor", *International Symposium on Computer Architecture (ISCA)*, 2004