A Fast and Accurate Analytical Technique to Compute the AVF of Sequential Bits in a Processor

Steven Raasch
Intel
Hudson, MA
steven.e.raasch@intel.com

Arijit Biswas
Intel
Hudson, MA
arijit.biswas@intel.com

Jon Stephan
Intel
Hudson, MA
jon.stephan@intel.com

Paul Racunas
Nvidia
Westford, MA
pracunas@nvidia.com

Joel Emer
Nvidia / MIT
Westford, MA / Cambridge, MA
emer@csail.mit.edu

Abstract

The rate of particle induced soft errors in a processor increases in proportion to the number of bits. This soft error rate (SER) can limit the performance of a system by placing an effective limit on the number of cores, nodes or clusters. The vulnerability of bits in a processor to soft errors can be represented by their architectural vulnerability factor (AVF), defined as the probability that a bit corruption results in a user-visible error. Analytical models such as architecturally correct execution (ACE) lifetime analysis enable AVF estimation at high speed by operating at a level of abstraction well above that of RTL. However, sequential elements do not lend themselves to this type of analysis because these bits are not typically included in the abstracted ACE model. Brute force methods, such as statistical fault injection (SFI), enable register level detail but at the expense of computation speed. We have developed a novel approach that marries the computational speed of the analytical approach with the level of detail of the brute force approach. Our methodology introduces the concept of "port AVFs" computed by ACE analysis on a performance model and applies these values to a node graph extracted from RTL. We employ rules derived from set theory that let us propagate these port AVFs throughout the node graph using an iterative relaxation technique. This enables us to generate statistically significant AVFs for all sequential nodes in a given processor design in a fast and accurate manner. We use this approach to compute the sequential AVF for all nodes in a modern microprocessor and show good correlation with beam test measurements on silicon.

Categories & Subject Descriptors

B.8 [Performance and Reliability]: B.8.1 Reliability, Testing, and Fault-Tolerance

Keywords

Reliability, soft error, sequentials, fault injection, fault simulation, AVF, ACE analysis

1 Introduction

Soft errors induced by alpha particles from packaging or from atmospheric neutrons are a significant source of transient errors in modern microprocessors. The rate of occurrence of these *soft errors* increases as bit counts increase, posing a significant risk for multi-core processors and systems-on-chip. Soft errors have become a core-limiting problem in modern multi-core server processors used for data centers, high performance computing, and other safety-conscious and mission critical compute segments. Accurate characterization of these errors is required to effectively deploy mitigation techniques.

Product soft error rates (SER) are generally given in terms of FIT or 'Failures In Time' (1 FIT equals 1 failure in 1 billion hours). Equation 1 shows that the FIT rate for a structure is a product of the intrinsic error rate of a bit in a storage structure (determined by the physical and electrical characteristics of the manufacturing process and circuit topologies), the total number of bits in the structure, and the architectural vulnerability factor (AVF). The ability to accurately compute each of the components of the FIT equation is essential to computing an accurate SER.

Architectural vulnerability factor (AVF) is the probability that a fault in a bit becomes a user-visible error. AVF has a significant impact on the overall SER. As a result, tools and methodologies to compute accurate AVFs for various components of the processor have been developed by both industry and academia. The addition of Architecturally

$$\text{SER FIT} = (\text{AVF}_{\text{bit}}) \times (\#\ \text{Bits}) \times (\text{Intrinsic Error Rate}_{\text{bit}})$$

Equation 1: Error Rate (FIT) Calculation

Correct Execution (ACE) modeling into performance models has proven very effective at computing statistically significant AVFs for high-level micro-architectural array structures [1][2]. Determination of AVF is a statistical process that requires the simulation of hundreds to thousands of workloads for tens of millions of instructions. Accurate assessment of the overall AVF of a microprocessor requires comprehension of the AVF not only of arrays, but also flops and latches (referred to as *sequentials*), and combinational logic.

While ACE analysis is largely limited to array structures, Statistical Fault Injection (SFI) into register transfer level (RTL) models can compute AVFs for sequentials, but at a high computation cost [7]. Due to the required number of workloads and instructions, the low speed of RTL simulation has prevented computation of high-quality AVF values for sequentials. RTL emulation provides some relief due to the substantial speedup this enables, but emulation can be costly to set up and maintain and still falls short of the speedup needed to provide statistically significant AVFs across multiple workloads for every sequential node in a processor. The technique described in this paper addresses this gap.

There are essentially two types of SER that are computed. One is silent data corruption (SDC), which measures the SER of components that do not have SER detection or mitigation. The second is detected uncorrectable error (DUE), which measures the SER of components that have error detection capability such as arrays protected with parity. In some instances, a third category of SER type that may be computed is detected corrected errors (DCE) which is the rate of errors on structures that have both detection and correction capabilities. The first two types however, are the most critical from a design, mitigation and specification standpoint.

In a typical modern microprocessor from Intel, about half of the processor's total SDC SER comes from sequentials [4]. In addition, as more and more register files and arrays are protected by techniques such as parity and ECC, the relative SDC SER contribution of sequentials will continue to increase even as the absolute SDC SER of the entire part decreases. Sequentials are much more difficult to protect than arrays with schemes such as parity. Instead, circuit techniques such as SEUT [3], BISER [4] or other Low-SER circuits [5] [6] are used to reduce the intrinsic FIT rate of the bits. Other methods include providing end-to-end protection [10] [11]. All of these approaches can introduce costs in terms of power, area, and performance. Another growing source of SDC is combinational logic. This logic most often is driven by and/or feeds sequentials. Hence, an accurate sequential AVF is also helpful in determining the true vulnerability of these combinational logic paths.

A fast and accurate means of determining the most vulnerable sequentials is required to determine the most efficient use of low-SER circuit and other SER mitigation techniques for these bits. In order to compute accurate AVFs for sequentials we need a technique that incorporates the level of detail of the RTL model with the simulation speed of the performance model.

We introduce a technique that accomplishes this goal. We have developed a tool flow based on this methodology that uses *port AVF* values (generated from ACE analysis in the performance model) with a node graph extracted from the RTL model. We use this flow to generate statistically significant AVFs for all sequentials in the RTL model in a fast and accurate manner. Computation times for our technique are on the order of a week to compute the AVF over thousands of workloads for 10s of millions of instructions for a modern processor core such as an Intel Xeon® CPU core. Additionally, our technique does not use statistical sampling, but actually generates AVFs for each and every functional sequential in the entire design in a single run.

We use a modern Intel Xeon® CPU core as our proof of concept, computing an average sequential AVF of 14% for a large variety of workloads. We also show ~66% improvement to the overall SER FIT correlation between modeled and measured values as a result of applying the sequential AVFs.

This paper makes the following contributions:

- Describes a novel methodology to compute statistically significant AVFs for all sequentials in a design by combining ACE analysis from a performance model with detail from RTL. This allows AVF computation for all sequentials in a design without the need for RTL level simulations.
- Develops and details a tool flow to implement the methodology, and describes the challenges at each stage.
- Describes the results of executing our tool flow on an Intel Xeon® CPU core as a proof of concept. We show, for the first time, accurate AVFs for sequentials across thousands of traces and tens of millions of simulation cycles. We show that these AVFs result in a significantly better correlation with silicon measurements for a few tested workloads.

This paper is organized as follows: Section 2 discusses a selection of related prior work in this area. Section 3 provides an overview of the two main established techniques for AVF computation and explains why neither is adequate for computing effective sequential AVFs as-is. Section 4 describes our novel sequential AVF computation methodology and the key concepts that enable it. Section 5 describes the implementation of this methodology as well as the tool flow. Section 6 shows the results of executing this tool flow using the Intel Xeon® core model as a proof-of-concept. Section 7 concludes this paper.

2 Related Work

There is a substantial body of related work around error modeling and the determination of AVF, We briefly review a few of the most relevant of those prior works here and show that none of the previous work addresses the problem being solved by this work.

The prior work can be categorized into two areas: on-line techniques and off-line techniques

2.1 On-Line Error Analysis

Two works that represent on-line techniques include the work by Li, et al. [17] that proposes adding physical design features to silicon in order to estimate AVFs on-line and Quantized AVF [20]. Unlike this work, these techniques can only provide AVFs for aggregate portions of a chip, not individual flops and latches.

These techniques are very different from accurate simulation of AVFs prior to silicon availability. Adding hardware for AVF estimation is a costly prospect for a commercial processor. Hence, accurate modeling, backed up by empirical testing, is essential. Our work specifically targets off-line analysis during the pre-silicon stage of design since the main idea is to identify SER vulnerable portions of the design to target for mitigation later in the design process.

2.2 Off-Line Error Analysis

Our work falls into the category of off-line techniques, which targets analysis during the pre-silicon stage of design. This category be further divided into pure SER characterizations, architectural modeling of soft errors, and techniques to apply fault injection methods.

There are several works that involve characterization of SER for low level circuits including work by Asad, et al. which builds on their previous work that uses signal probabilities to estimate SER rates [12]. They derive the probability of a “system failure” at some future time based on the fault location and the probability of a system failure ever occurring because of a fault in that location. Holcomb, et al performed circuit simulation over fixed workloads [13]. This work characterizes electrical, timing, and logical masking only. Both these techniques provide no sense of AVF, which is important for understanding the engineering tradeoffs for reducing SDC.

Architectural techniques, such as ACE analysis, can be applied at the pre-silicon design stage. For sequential AVF, however, we need detail that only the RTL-level model provides in order to compute the AVF for every individual state bit, and ACE analysis uses a performance model that does not provide that level of detail. The SoftArch technique developed by Li, et al. [16] is another architectural level tool for modeling soft errors. While SoftArch techniques could be applied at the RTL level, it would likely result in excessive runtimes that our technique avoids.

Much of the most relevant related work focuses on fault injection as the key technique to compute AVFs for flops and latches. Fault injection at the RTL level is extremely time consuming and much of the work to compute AVFs for flops and latches revolves around enhancing the speed of fault injection. Cho, et al. focused on the need to do fault injection at a sufficiently low level of detail such that flops/latches exist and can be characterized [19]. Saggese, et al. computed AVFs via fault injection into two workloads [14]. Blome, et al. performed fault injection into RTL for an actual ARM core [15]. Their analysis of error fanout and propagation behavior allowed them to derive logical and temporal masking rates. Hari, et al developed the GangES error simulation technique [18] to speed up fault injection by nearly 2x. While these techniques claim significant speedups for fault injection, they are still 3 to 4 orders of magnitude slower than what is required for a general-purpose commercial processor.

Our technique operates at a similar level of detail as fault injection (RTL model) but instead of injecting faults, we extend a well-known analytical technique in a novel new way to compute the AVFs of low-level structures, enabling a substantially faster and more comprehensive analysis. Our analytical method can compute the AVFs of thousands of workloads spanning tens of millions of cycles of execution for each workload in the course of hours to days. We show that there is no need to choose a low or high level of abstraction but can marry the strengths of both.

3 Established AVF Computation Techniques

There are two main accepted techniques to compute AVF pre-silicon. The first method is statistical fault injection (SFI) into an RTL model. The second method is known as ACE lifetime analysis (which is performed on an architectural performance model). Neither technique is adequate as-is to compute accurate, statistically significant AVFs for sequentials. Our technique therefore combines the best aspects of both: the analytical approach and simulation speed of ACE analysis with the level of detail (primarily state and connectivity information) of the RTL model.

3.1 SFI on RTL

Statistical fault injection, or SFI, works by running two copies of the RTL simulation. A fault is injected into one copy by artificially flipping a random bit at a random time-step. The simulations are then run for some number of cycles, usually 10,000 to 50,000. If a state mismatch occurs at a point that impacts correct program operation, the fault is considered to have propagated to an error. Faults resulting in errors contribute toward the AVF for that node. This process is repeated by injecting faults across a large number of state nodes and cycles. The sequential AVF is computed as the number of errors seen at the observation points divided by the number of injected faults. There is also an additional unknown component that is a result of injected faults that may still be resident in the system but have not propagated to the observation points by the time the simulation ends. The quality of the results depends directly on the number of injections that can be simulated and the number of cycles after the injection in which faults can propagate to the observation points. Equation 2 gives the AVF formula for the SFI technique.

RTL models are very slow due to the high level of detail. This lack of speed means that long simulations cannot be

$$\text{Sequential AVF} = (\text{\# Errors} + \text{\# Unknown}) / \text{\# Injected}$$

Equation 2: Calculating Sequential AVF using SFI

completed within a reasonable amount of time. Sequential AVFs can be computed using many short simulations (since the lifetime of data in a pipeline latch or flop is generally short), but this can increase the size of the unknown component significantly.

The AVFs for SDC and DUE must be computed separately, since the observability points for faults will be different. For SDC, the observability points are at the program outputs, while for DUE those points occur at the error detection logic for the structure/latch. Determining both SDC and DUE AVF potentially doubles the number of simulations, making the SFI technique even more costly.

As an example, a processor with 100,000 sequentials running a 10,000 cycle simulation would require 1,000,000 RTL simulations to inject into every potential fault for complete coverage of the solution space (100,000 sequentials x 10,000 cycles). Usually only a small sample set of the solution space is simulated. Appropriate guardbands are then applied to compensate for the partial coverage. However, to perform this operation for even a dozen or so workloads can easily result in more than a billion simulations to achieve statistically significant AVFs for those workloads. Typically, this can take months to years of simulation time on a modern microprocessor for just a few workloads.

For these reasons, computing sequential AVFs using SFI into RTL is not realistically feasible for any modern design with millions of sequentials or for hundreds to thousands of workloads. Although the level of detail lends itself to computing sequential AVF well, the compute cost of doing so is prohibitive. We should point out however, that SFI into RTL is still the best way to compute limited AVFs for a handful of data or control paths on a few specific workloads. Since all state and masking effects are fully modeled, this is appropriate either to validate analytically modeled results or to compute precise AVFs for very specific conditions.

3.2 ACE Lifetime Analysis on Performance Models

ACE Lifetime analysis [1] [2] is an analytical method to compute AVF values for processor storage structures. ACE stands for architecturally correct execution and introduces the notions of ACE (necessary for architecturally correct execution) and un-ACE (un-necessary for architecturally correct execution) instructions and data. ACE lifetime analysis and hamming-distance-1 analysis [2] are performed using an ACE-instrumented performance model.

ACE lifetime analysis is a completely analytical technique that does not rely on fault injection. ACE analysis monitors read/write events to compute the residency time of ACE bits in a structure during that structure's lifetime, where *structure* refers to micro-architectural storage elements such as buffers, register files, queues, caches and other types of storage arrays. The AVF for a structure is calculated by

$$\text{Structure AVF} = \frac{\sum \text{residence time of all ACE+unknown bits in a structure}}{(\text{\# bits in structure}) \times (\text{total simulation cycles})}$$

Equation 3: Calculating Structure AVF Using ACE Analysis

dividing the average ACE lifetime for all bits in the structure by the total simulation time. This technique is extremely well suited for computing structure AVFs since structure behavior must be tracked over long periods of time due to potentially long fault latencies.

To compute structure AVFs, one copy of the model is run for each benchmark for tens of millions of cycles. Since the performance model is 100X-1000X faster than RTL, it is possible to run hundreds to thousands of traces for tens of millions of cycles in just a few days or weeks. This allows for a very robust and broad sampling of the average behavior of any particular structure. It also allows the structure AVFs to be targeted to specific workloads and/or application suites. The final structure AVF is simply the fraction of cycles that the structure contains ACE state (see Equation 3).

Using this method, SDC and DUE AVFs can be computed in a single run. Disadvantages of this methodology are primarily associated with the fact that performance models do not have a high level of detail. As a result, only structures that exist in the performance model can be analyzed and their AVF computed. Many high AVF structures not required for typical performance studies, such as microcode registers, scratchpads, and control registers, and therefore may not be modeled. Additionally, none of the millions of sequentials that make up the random logic state, the data path and control path pipelines, nor any of the staging logic found in modern processors is modeled in sufficient detail to compute AVFs.

ACE analysis has been used effectively on recent processor designs to compute the AVFs for most high-level micro-architectural structures. While specific sequentials and data paths can be modeled in a performance model and have AVFs computed, to do so for all sequentials would render the performance model no faster than the RTL model, thus defeating the purpose of having a higher level model.

Due to the reasons stated in this section, while ACE analysis provides a more analytical approach to computing AVF than SFI, it does so using a performance model. The performance model must be used to provide enough simulation speed to compute AVFs across enough workloads and cycles to be statistically significant. This precludes the ability to provide the low-level detail necessary to model all the millions of sequentials in the actual design.

4 Computation Methodology

Our technique to compute Sequential AVF applies a hybrid approach that uses data obtained from the ACE analysis in the performance model along with detailed signal flow information taken from the RTL design. ACE data rates from the performance model are propagated through the RTL node graph to obtain AVF information for each node. This approach allows us to obtain RTL-level detail without having to run lengthy RTL simulations.

The key piece of data required is the *port AVF* (pAVF) of each ACE-evaluated structure. The pAVF of a bit in a structure's port or interface is the probability that ACE data will be transmitted to or from the structure through that bit. For a read port, $pAVF_R$ is calculated by dividing the number of ACE reads from the structure by the total number of cycles simulated. For a write port, we divide the number of ACE writes to the structure by the number of simulated cycles to compute $pAVF_W$. The ACE-read and ACE-write count values for each structure are reported by the ACE model.

These values represent the ACE data rate of the circuits immediately adjacent to the structure (essentially the logic that represents the read/write ports of the structure). While both structure AVF and pAVF measure the same characteristic of a bit, we use pAVF for this discussion to avoid confusion with structural AVF as well as to describe how the value is calculated. A pAVF value can be calculated for all nodes (both combinational and sequential) in a design.

It is important to note that when discussing an ACE structure within the RTL, we are referring only to the set of storage elements used to hold structure values and not to the logic associated with those elements. AVF can be approximated under certain conditions using Little's Law. Prior work has shown that AVF can be computed as the product of the average ACE latency and the average ACE throughput [1] [2]. One major difference between array structure AVFs and port AVFs is that the array structures' AVF is usually dominated by ACE latency while the AVF of the ports are dominated by the ACE throughput.

The remainder of this section describes the details of the methodology used to calculate the AVF of these sequential circuits. As discussed above, this methodology involves propagating pAVF values through a node graph generated from RTL. At each step during this propagation, the pAVF estimate for the current node is computed and the node is annotated with that value.

In order to limit the scope of the initial implementation of the sequential AVF methodology, it was necessary to make three simplifying assumptions. They are made in such a way as to ensure that our final results are conservative. Several of these assumptions provide an opportunity to refine the estimate if one is able to use additional information to perform the analysis.

First, we note that values that are stored for more than one cycle makes it impossible to reason about the ACE data rates around that storage element. Since we are focusing on the sequentials such as those found in pipeline latches, our analysis assumes that any data stored for more than one cycle is stored in an ACE-instrumented structure or an identifiable control register. Our implementation automatically identifies most control registers and treats them as ACE structures with 100% read-port pAVF ($pAVF_R$). Similarly, our implementation identifies sequentials that behave as ACE structures (data is read/written via enable/enabled clock signals) and these are then modeled in the ACE performance model.

Second, since our analysis does not simulate the logic represented by the RTL, we are not able to identify when logical masking of ACE values occurs. We conservatively assume that there is no logical masking beyond the microarchitectural-level logical masking analysis already accounted for in the ACE model.

Third, we assume that most design debug and instrumentation logic in production RTL that do not play a role in normal product operation are eliminated from the processed RTL either by compilation settings or by the sequential AVF computation tool via naming conventions. We intentionally leave certain debug control code in place since faults in these nodes can result in improper operation of the final product.

4.1 Propagating pAVF Values

The process of propagating pAVF values through RTL involves taking each known pAVF value and traversing the associated RTL node graph. To simplify this discussion we will split these traversals into two phases: walks propagating down from structure read-ports (forward propagation of $pAVF_R$), and walks propagating up from structure write-ports (backward propagation of $pAVF_W$). A walk is terminated when all of its paths have reached an ACE structure, an RTL boundary, or a node already visited during this walk.

A walk terminates when it reaches a structure because the pAVF values for the structure are measured values that will be more accurate than the estimated values arriving via the walk (in this way ACE-modeled structures are treated as "sources" and "sinks"). In the case where a walk reaches a node that it has already visited (the walk has encountered a loop), continuing the walk does not add new information, and can be terminated.

During this discussion, it is important to remember that the flow of ACE bits is independent of the logic function within the circuit. While a circuit element may be depicted as a NOR gate, the function is not of consequence. Further, care must be taken to avoid thinking in terms of a bits '1' or '0' state, and remember that the only factor to consider is whether the bits being discussed are ACE or un-ACE. It is also useful to think of the pAVF values as representing the ACE data-rate.

4.1.1 Forward Propagation of $pAVF_R$

We begin this discussion with the observation that any data path can be broken down into a set of interconnected subsections made up of simple pipelines, logical join points (fan-ins), and distribution split points (fan-outs). This section will discuss how pAVF values are propagated through each of these topologies, and how signals are annotated along the way. First, we will discuss the general approach to propagating pAVF values from structure outputs to structure inputs.

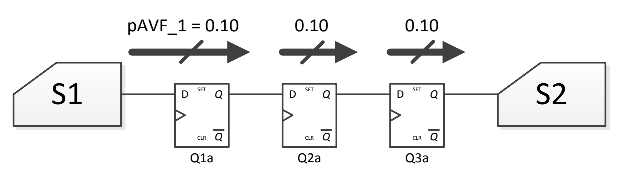

Figure 1: pAVF simple pipeline propagation

$$\text{AVF}(Q1a) = \text{AVF}(Q2a) = \text{AVF}(Q3a) = \text{pAVF}_R(S1)$$

Equation 4: Forward Sequential AVF of Simple Pipeline

$$\text{AVF}(Q1a) = \text{pAVF}_R(S1)$$
$$\text{AVF}(Q1b) = \text{pAVF}_R(S2)$$
$$\text{AVF}(Q2a) = \text{pAVF}_R(S1) \cup \text{pAVF}_R(S2)$$
$$\rightarrow \text{AVF}(Q2a) = \text{pAVF}_R(S1) + \text{pAVF}_R(S2)$$

Equation 5: Forward Sequential AVF of Logical Join

$$\text{AVF}(Q1a) = \text{AVF}(Q2a) = \text{AVF}(Q2b) = \text{pAVF}_R(S1)$$

Equation 6: Forward Sequential AVF of Distribution Split

The simplest case is that the path between two structures with known port AVFs is a straight-line series of pipeline sequentials with no logical joins or distribution splits. This case is illustrated in Figure 1. The figure shows a structure S1, for which a pAVF$_R$ has been calculated by performance model analysis. A second structure S2 is connected to S1 by a series of sequentials. We wish to calculate the AVF of each of the intervening sequentials. In this example, each value that enters the pipeline from the read port of structure S1 will eventually enter the write port of structure S2. Any ACE value entering the pipeline will still be ACE at the pipeline output. Thus, each sequential in this simple pipeline will have the same AVF value, equivalent to the pAVF of structure S1's read port given by Equation 4.

Figure 2 shows a circuit containing a logical join point. Here the read ports of two evaluated structures, S1 and S2 feed a circuit that drives the write port of S3. The NOR gate G1 combines the output signal of S1 with the output signal of S2. To determine whether the output of G1 is ACE, we consider the possible combinations, focusing on a single cycle. One possibility is that the values held in Q1a and Q1b are both unACE. In this case, we can be certain that the output of G1 is also unACE, as a value that is not necessary for architecturally correct execution cannot create a result that is necessary for correct execution in a single fault model. A second possibility is that Q1a holds an ACE value while Q1b holds an unACE value, or vice versa. When these values are combined, the result may or may not be ACE. In this case, we make the conservative assumption that the result is also ACE. The final possibility is that both Q1a and Q1b hold an ACE value. In this case, the output of G1 will be ACE. The pAVF of the output of a logical join point (Q2a in Figure 2) is conservatively considered to be the union of the pAVF values of the inputs to the join point. If we further assume that there is no overlap between the pAVFs of S1 and S2 then the union

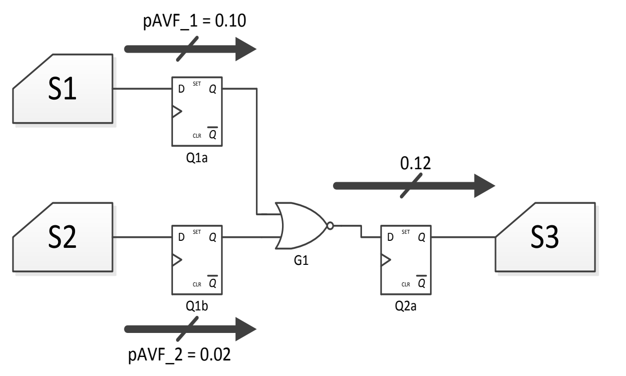

Figure 2: pAVF logical join propagation

Node pAVF starts at 1.0
New Node pAVF = MIN(Node pAVF, Walk pAVF)

Equation 7: Node Update Rule

simplifies to the sum of the pAVF$_R$ of S1 and S2. This is shown in Equation 5.

Figure 3 shows a circuit containing a distribution split point. Here the read port of one evaluated structure, S1, feeds a branching circuit that ends at the write ports of S2 and S3. Since the values held in sequentials Q2a and Q2b will always be generated from the value in Q1a, the AVFs of all three sequentials are equal as shown in Equation 6.

The process of estimating node pAVF values requires that the known pAVF values from each ACE-instrumented structure be propagated throughout the RTL. We begin the process by examining the known pAVF values of the structure outputs and walk these values through the RTL node-graph, annotating each visited node with the appropriate pAVF value. Since each bit of each structure will have different connectivity, it is necessary to perform the node-walk for each bit of the structure.

Each pAVF walk continues until it encounters either the boundary of the RTL under analysis or a structure input. Since each structure has a known AVF and pAVF$_W$, the portion of the walk that encounters the structure will terminate at that point. A walk will also stop when it encounters a node that it has already visited, since continuing would be redundant. An advantage of this approach is that loops in the node-graph (e.g. from an FSM) are automatically broken.

All nodes conservatively start with a pAVF of 1.0. As the walk visits each node, the walk pAVF value is compared to the node's current pAVF. If the walk pAVF value is less than the node pAVF, then the node is updated to use the walk pAVF. This is shown in Equation 7.

4.1.2 Backward Propagation of pAVF$_W$

Note that while the structure pAVF$_R$ (Read port pAVF) values we have annotated are conservative estimates, the structure pAVF$_W$ (Write port pAVF) values are available to

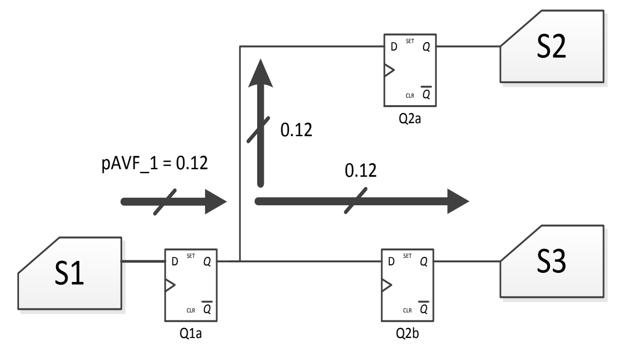

Figure 3: pAVF distribution split propagation

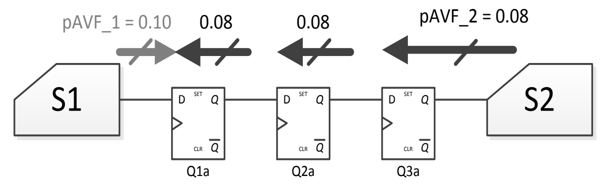

Figure 4: pAVF propagating up through a simple pipeline

refine the results. We know that (because each node estimate is conservatively based on the previous node) the farther the walk is from the source of the pAVF value, the less likely it is to be accurate. Walking backwards through the node-graph from a structure's write-port the average distance between an annotated node and the source of its pAVF value will be reduced, increasing the overall accuracy of the estimate. We refer to the walk from structure inputs to outputs as a walk "up" the node graph.

Remember that what we are propagating is essentially a signal probability (the probability of an ACE bit instead of the probability of a one or zero), not a data item. Given this, we can reason about the pAVF of a set of inputs given the output of a circuit. We break down the RTL into the three topologies discussed earlier.

Referring to Figure 4, the simple pipeline does not create or destroy ACE data, so the pAVF of the pipeline input must be equal to the pAVF of the output. It should be noted that since S1's read port pAVF is computed independently of S2's write port pAVF in the performance model and both estimates are conservative, their calculated values may not necessarily match. The backward AVF equation is shown in Equation 8.

Walking up through a logical join operation (Figure 5) requires that we determine the pAVF of two or more inputs from the pAVF of the output. Regardless of the operation performed, the worst-case pAVF of an input can be no larger than the pAVF of the output, so we conservatively assign the output pAVF value to both input signals. This is given by Equation 9.

For a distribution split point where there are likely two or more different pAVF values at the outputs as shown in Figure 6, as with the case of propagating down through a logical join

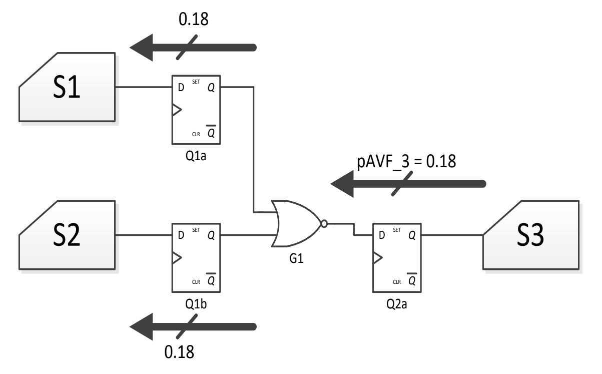

Figure 5: pAVF propagation up a logical join

$$AVF(Q1a) = AVF(Q2a) = AVF(Q3a) = pAVF_W(S2))$$

Equation 8: Backward Sequential AVF of Simple Pipeline

$$AVF(Q1a) = AVF(Q1b) = AVF(Q2a) = pAVF_W(S2)$$

Equation 9: Backward Sequential AVF of Logical Join

$$AVF(Q2a) = pAVF_W(S2)$$
$$AVF(Q2b) = pAVF_W(S3)$$
$$AVF(Q1a) = pAVF_W(S2) \cup pAVF_W(S3)$$
$$\rightarrow AVF(Q1a) = pAVF_W(S2) + pAVF_W(S3)$$

Equation 10: Backward Sequential AVF of Distribution Split

point, we do not know the relationship between the two pAVF values. Therefore, we conservatively assign the union of output pAVF values to the distribution point's input. In Figure 6, we again assume that there is no overlap in the $pAVF_W$ for S2 and S3 so the union is simply a sum capped at 1.0. This is given in Equation 10.

These rules are all that is required to propagate pAVF values through the circuits being analyzed. We proceed to walk pAVF vales from each bit of each structure's write-port. The walks update each visited node, annotating the node with the calculated pAVF value, until it reaches an ACE-structure, an RTL boundary, or a node that it has already visited. Since each walk makes use of only the assignment and addition operations, the walks can be done in any order.

4.2 Final pAVF Computation and Propagation Example

After completing both the "up" and "down" walks, most nodes are annotated with two pAVF values. For the nodes that have pAVF values computed by the ACE model, the estimate value is discarded in favor of the computed value. For the remaining nodes, the smaller of the two estimates can be used since both values are obtained conservatively. The fact that we can use the minimum of the two values is a key point in this technique and is the main reason why the node AVF values do not simply saturate to 100%.

Table 1 shows the final AVF equations for the node types based on reconciling the pAVF values for the previous examples.

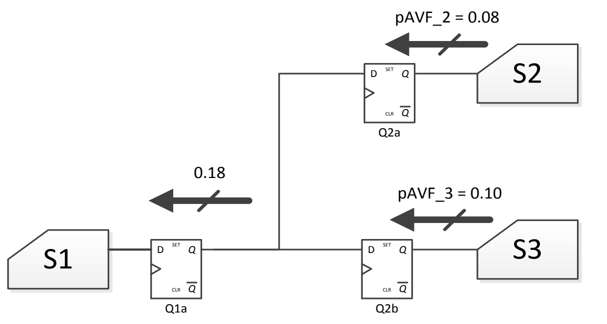

Figure 6: pAVF propagation up distribution split point

Node Type	Final Sequential AVF Equation
Simple Pipe	AVF(all nodes) = MIN(pAVF$_R$(S1), pAVF$_W$(S2))

Logical Join	AVF(Q1a) = MIN(pAVF$_R$(S1), pAVF$_W$(S3))
	AVF(Q1b) = MIN(pAVF$_R$(S2), pAVF$_W$(S3))
	AVF(Q2a) = MIN((pAVF$_R$(S1) + pAVF$_R$(S2)), pAVF$_W$(S3))

Distribution Split	AVF(Q2a) = MIN(pAVF$_R$(S1), pAVF$_W$(S2))
	AVF(Q2b) = MIN(pAVF$_R$(S1), pAVF$_W$(S3))
	AVF(Q1a) = MIN(pAVF$_R$(S1), (pAVF$_W$(S2) + pAVF$_W$(S3)))

Table 1: Final Sequential AVF Equations for All Node Types

The example shown in Figure 7 illustrates the pAVF propagation process based on the rules in Table 1. Blocks S1-S4 represent ACE structures. Note that the pAVF values for the structure read and write port signals are provided in the figure (in practice these values are computed from the ACE model).

The first phase of the pAVF walk begins with the walk from the S1 read-port (pAVF_1). This pAVF is applied to the output of Q1a and Q2a. Both of these signals are annotated with 0.10 since they form a simple pipeline. The pAVF for G1 and G2 cannot be determined without further information, so the walk ends here.

Next, the S2 read-port pAVF (pAVF_2) is walked forward to the output of Q1b, which is annotated with 0.02. Gate G1 forms a logical join-point, so its output is annotated as the union of the pAVF$_R$ values from S1 and S2 (pAVF_1 U pAVF_2). Remembering that we calculate the sum of pAVF values for the union, the output is annotated with a pAVF value of 0.12. This value is then propagated forward through Q3b. Additionally, this value is applied to the second input of gate G2.

Gate G2 forms a logical join-point, with input values pAVF_1 and (pAVF_1 U pAVF_2). The union of these values is (pAVF_1 U (pAVF_1 U pAVF_2)), which simplifies to just (pAVF_1 U pAVF_2). As before, the result of the union is applied to the output. The outputs of both G2 and Q3a are annotated with a pAVF value of 0.12 (0.10 + 0.02). This concludes the first phase of the pAVF walk.

The second phase of the pAVF walk begins with the write-port pAVF values for structures S3 and S4, and continues until the walks reach S1 and S2. The mechanics of these walks are identical to the first phase walks, with the

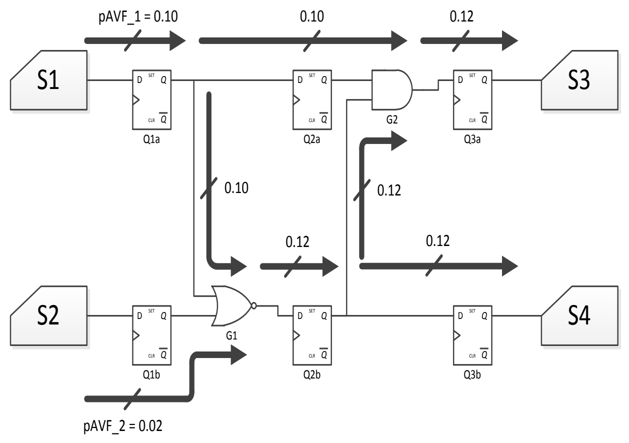

Figure 7: Circuit for pAVF Propagation Example
Arrows indicate forward propagation of pAVF values from S1 and S2.

exception that the computation for logical join points and distribution split points are performed as discussed in Section 3.1.2.

The values are then resolved using the rules in Table 1 resulting in a final AVF for each sequential.

4.3 Accounting for Loops

One of the key challenges we faced was how to deal with loops, such as those created by state machine feedback paths. Examples of such loops are abundant in modern microprocessors including stall loops, head and tail pointer update loops and so forth.

Loops, even though they are made from sequentials, behave like structures. That is, they can retain state and therefore pAVF values cannot simply propagate through them. Depending on the actual logic in the loop, values can get “stuck”, remaining resident and breaking our 1-cycle latency assumption. Therefore, loops can be dominated by latency more than throughput, just like the structures modeled in the ACE model.

A few different solutions exist to deal with loops.

1. They can be modeled in the ACE model and treated as structures. However, this requires the ability to extract all loops from the RTL and model them in the performance model. This is not a simple task as some loops can encompass dozens of pipe stages and even incorporate other structures and even other loops (nested loops).
2. RTL simulations can determine the probability of loops retaining values versus passing values. This probability can be the pAVF for the loop. Again, loop identification can be an issue, especially with nested loops. However, this defeats the purpose of our technique by requiring RTL simulations.
3. Assume some static pAVF value for the loops. The RTL node walker can easily find and break loops and inject static pAVF values into those nodes. Effectively this treats the loop nodes as a structure

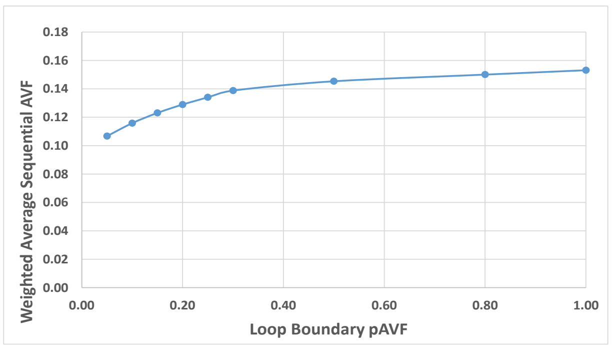

Figure 8: Avg. Seq. AVF for different Loop Boundary pAVFs

and pAVF walks will start and stop at these nodes. The challenge is in choosing a static value that is conservative without causing the propagated pAVFs to saturate to some very high value.

Solution 3 is clearly the simplest solution to implement. However, the challenge is that, in order to be conservative, the static value would likely need to be 100%. This may cause the resulting sequential AVFs to be pessimistically conservative. However, our studies have shown that there is relatively little variation in the resulting average sequential AVF across the entire design (see Figure 8) for different values of loop-boundary pAVF. Interestingly, a 100% pAVF applied to every loop boundary node did not cause the sequential AVFs to saturate, nor was the effect linear. Lower points showed a modest decrease but there appears to be a heel in the curve around 30%. This value correlates well with the typical conservative AVF value we derived from our work on structure AVFs. These loop-structures can clearly not be treated as simple logic, nor can they be analyzed as a conventional storage structure.

Based on the results of the studies show in figure 8 for different loop-boundary values, we chose 0.3 as an appropriate value for our analysis.

Only about 2%-3% of sequentials are in loops. However, the problem stems from the fact that the AVF used for loops could have a ripple effect and propagate into sequentials fed by, but not part of, the loop. Hence, deriving a less conservative number for loop sequentials helps reduce the overall conservatism.

The expectation is that other designs would behave in a similar fashion. The other pAVFs as well as the MIN functions do a very effective job keeping the AVFs from saturating to 100%. Additionally, the complexities of the node graph itself and the relaxation approach help refine high AVFs even further. Other designs may have the heel of the curve at a different point, but this is a simple study to run for each design. Once chosen, this loop node pAVF value can be applied to all such nodes.

Most applications of AVF look at higher granularities than individual flops and latches. Usually they will be used to target a functional block or data/control path. In that case the law of averages will help smooth out perturbations introduced by this sort of approach. If a deeper level of accuracy is required, then one of the other 2 approaches may be considered on a case by case basis.

5 Implementation and Tool Flow

This section describes the tool flow we developed to integrate sequential AVF computation into our existing RTL design flow. The tool flow includes a number of individual programs involved in RTL translation, pAVF propagation, intermediate data handling, and tool automation. The full analysis process can be broken into four major steps:

1. Develop ACE model on detailed microarchitectural performance model
2. Collect pAVF data from ACE model
3. Compile RTL
4. Map ACE structure bits to RTL bit names
5. Walk pAVF values through RTL

5.1 Tool Development and Flow

Most existing product design flows for modern processors already include the development of a detailed micro-architectural performance model as well as RTL. Our tool flow leverages the existing collateral in order to simplify implementation.

The first step in this regard is to develop an ACE model on top of any existing performance model. This may include adding additional detail into the performance model to support the ACE modeling effort. Our ACE model includes the standard ACE Lifetime analysis as detailed by Mukherjee, et al [1] as well as the Hamming-Distance-1 analysis for address based structures as detailed by Biswas, et al. [2].

Additionally, we noted that many structures, especially control structures, tended to hold bits that were used in different ways. This was exhibited by an entry being broken into various bit fields representing different pre-coded information. Not all the bit fields were ACE simultaneously, but rather depended on the instruction, data type, or other micro-architectural details. As a result, we modeled each bit field of these structures as a separate ACE structure. This process, which we called *"Bit Field Analysis",* provided a much more detailed level of ACE analysis for these control structures than before. The resulting pAVFs can be much less conservative as a result.

The standard RTL compilation process was invoked to generate intermediate-format RTL files (called EXLIF files) necessary for further processing. This compilation should be invoked in such a way as to remove as much instrumentation and Debug-only (DFX) logic as possible, leaving only the logic that is susceptible to transient errors during normal operation. DFX logic that can generate run-time errors (e.g. debug-mode enables) will need to be processed, and should be retained. The output of the compilation process is a group of RTL module files that correspond almost exactly to design functional blocks (FUBs). After compilation, the EXLIF files are further processed by a new tool to fully expand each FUB module by instantiating all sub-circuits within that module. When complete, each EXLIF file contains a single model statement that represents the original FUB with all hierarchy removed.

The third step involved mapping between the high-level structures found in the ACE model and the actual bits in the RTL. Often an individual structure is composed of several arrays. It is not uncommon for some of the arrays to be contained in a different FUB. Each of these RTL bits has an associated $pAVF_R$ and $pAVF_W$ values in a text file that tells the next tool which RTL bits to treat as structure bits.

Circuits that lie outside of the RTL being analyzed are grouped together into one or more pseudo-structures, with its own $pAVF_R$ and $pAVF_W$ values.

The Sequential AVF Resolution Tool (SART) is the key tool in the tool chain and is responsible for walking pAVF values through the RTL in order to resolve the sequential AVF for each node. As discussed in section 4, the tool runs in three phases: (1) walking down from structure read-ports, (2) walking up from structure write-ports, and (3) the final resolution phase that chooses the estimated value from the first two phases. The relaxation process runs SART repeatedly until the values reach a sufficiently steady state.

SART attempts to identify configuration control-register bits, usually by the RTL name or the driving clock. These bits are assigned a $pAVF_R$ of 100%. Since writes to these control registers are relatively rare, the $pAVF_W$ will approach 0%. As a result, we can omit walks up from these write-ports.

5.2 RTL Challenges and Recommendations

RTL models for modern processors can be large, consisting of hundreds of FUBs. Processing the entire RTL model into a single node graph can be very expensive from a compute resource standpoint, resulting in very large memory footprints and slow node traversal as a result.

It may be advantageous to partition the RTL to be processed in order to better fit available computing resources or to parallelize the task. For our purposes, the natural boundaries of the RTL are at the FUB boundaries.

The challenge we faced is related to the fact that each RTL FUB cannot produce meaningful results without inputs from the surrounding FUBs. We chose to deal with this situation using a relaxation approach that calculates the AVF for the entire design repeatedly over several iterations, refining the AVF values each iteration.

The process requires that we maintain a global netlist of connections between the FUBs that is annotated with the pAVF values that need to be applied to each FUB (the FUBIO values). For each FUB output node, the node's $pAVF_R$ is collected to be applied to consuming input nodes in other FUBs. Similarly, for each FUB input node, the nodes $pAVF_W$ is collected for use by producing nodes in other FUBs. The same rule previously discussed for internal logic (smallest conservative value is used) is applied to the FUB interconnects. This "merge" step is completed at the end of each iteration, after all FUB's have been analyzed.

During subsequent analysis iterations (defined to be one up and one down walk through the netlist for each FUB), the merged FUBIO information is used as an input to the analysis. In this way, pAVF values traverse from one FUB to the next, adding their contribution to the nodes in the FUB, until they reach a terminating node.

When processing partitioned RTL, it is important to note that any walk can only cross one partition during each iteration. In order for an individual pAVF value to be visible on the partition boundary and then walk through a second partition, requires two SART iterations. For each subsequent walk through an adjacent partition requires another iteration.

As the number of iterations increases, the walked values will eventually propagate to their final termination points, and the change seen in pAVF values due to those values will drop to zero. The number of executed iterations will vary with the RTL being processed. A sequence of iterations can be terminated either by limiting the number of iterations or waiting for some measure of pAVF change to reach a threshold. For our RTL, we found that 20 iterations was sufficient to achieve convergence.

Another optimization made to the tool flow involved propagating the pAVF values symbolically through the RTL node graph. As the pAVFs propagate in this way, a closed form equation is generated for each visited node in the netlist with the terms of the equations being the structure pAVFs of the ACE model plus any injected state (such as from control registers or loop boundaries). Once the iterations complete and the equations are no longer changing, we are left with a closed form AVF equation for every node in the RTL netlist such that each node's AVF can be computed as a function of injected and ACE structure pAVFs.

The benefit of this is that any subsequent sequential AVF computations on this particular design simply needs to generate new pAVFs from the ACE model then plug those values into the closed form equations to compute new sequential AVFs. No subsequent sequential AVF computation needs to re-run the SART or relaxation stages of the tool flow, saving considerable effort.

6 Results

To test the proof-of-concept tool chain, we analyzed the RTL for an Intel Xeon® core processor design.

6.1 Experimental Setup and Modeling

We collected pAVF values from a set of 547 workloads from a custom server benchmark suite. The suite includes industry-standard benchmarks such as SPEC as well as traces of actual server workloads such as transaction processing, web benchmarks. The pAVF values were collected from over 100 ACE-modeled structures, and then mapped to over 200 latch arrays from the RTL. SART identified 6825 bits as being configuration control register bits. Each of these control registers was treated as a single structure with $pAVF_R$ of 100%. The tool also identified 201,530 bits belonging to

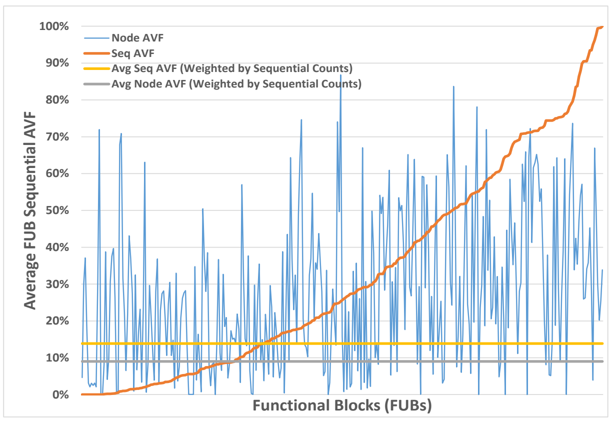

Figure 9: Per-FUB Average Sequential AVF Results

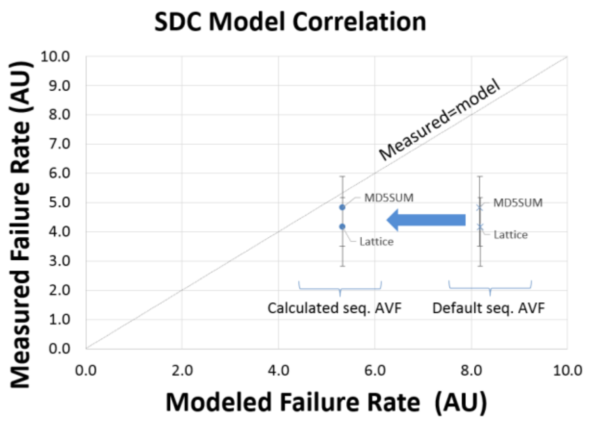

Figure 10: Beam test correlation results with model

loops. Loop AVF values were chosen as discussed in section 4.3.

The SART walks were executed with the design partitioned into individual RTL modules, and the appropriate FUBIO merging performed at the end of each iteration. The results presented here required 20 iterations, with intermediate data indicating that this was a sufficient number of iterations for convergence. We evaluated convergence here by plotting the average pAVF of sequentials for each FUB over each iteration.

The SART analysis required about a day to run, and visited more than 98% of all RTL nodes. Figure 9 plots the average FUB sequential AVF for each RTL module after the last iteration. The overall averages are weighted to account for the actual number of sequentials in each FUB, indicating that the vast majority of sequentials were actually in the lower AVF FUBs. As expected, most FUBs have significantly smaller pAVF values than the average structure AVF from the ACE model. This represents a ~10% reduction in overall modeled SDC FIT for the processor. For reference the weighted average of both sequential pAVF and node pAVF (representing combinatorial and sequential nodes) are plotted.

Note that for any individual FUB, there is little correlation between the total average node AVF and the average sequential node AVF. This is due to the different circuit configurations and the relative number of combinational nodes versus sequential nodes. For example, a high-AVF logic branch may have relatively few sequentials compared to a low-AVF branch.

6.2 Correlation with Silicon

The final determination of success for any modeling innovation is in how well it correlates (or improves correlation) with actual measurement.

We selected two workloads for which we have good SER data taken during accelerated SER testing of Intel Xeon® processors. The selected workloads were:

- Lattice – this workload calculates the location of a particle in a 3d lattice with inter-particle forces. We modified it to be a 2d lattice.
- MD5Sum – this workload calculates 128-bit MD5 hashes as per [9]. It was modified to remove memory accesses (to reduce cache DUE since memory errors can quickly overwhelm SDC observability under the beam), and therefore does not calculate a true MD5 hash, though it does all the same calculations.

The accelerated conditions were created at the Indiana University Cyclotron Facility using a 200 MeV proton beam with variable flux.

The measurement setup was the same as that used for prior work such as the SDC virus measurement testing [8], and using the Direct Memory Load infrastructure to minimize DUE.

Prior to the sequential AVF work, our model to measurement correlation for SDC was off by nearly 100% with the modeled SER being higher than the measured. In this case, we were conservatively using structure AVFs as a proxy for the sequential AVF. The expectation was that, since the sequential AVFs value should be dominated by throughput rather than latency, the modeled SER should decrease when we began computing sequential AVF values. While this provided us with a very conservative modeled estimate for the SDC SER, the level of miscorrelation made it difficult to justify design changes to mitigate SER without a more accurate correlation.

The new modeled sequential AVF for these benchmarks were 63% lower than the conservative values we had been using. As a result, the model/experimental correlation improved by ~66%, which is within the statistical error of the measured value, as shown in figure 10. Note that due to the sensitive nature of the actual FIT values we normalize the values to Arbitrary Units (AU).

7 Conclusion

We have demonstrated that it is possible to obtain a much tighter AVF bound for miscellaneous sequentials using a purely analytical technique. We presented a hybrid approach which uses structures' *"port AVFs"* computed by the performance model and applies them to an extracted node graph from RTL. This technique marries the detail level of the RTL with the speed of the high-level analytical model, providing a feasible way to compute accurate sequential AVF values. The values of millions of sequentials are calculated based on inputs from thousands of workloads in a very reasonable amount of time. We presented our tool flow implementation as well as detailed some of the specific challenges and recommendations that helped us simplify tool development and implementation.

We used this methodology and tool flow to generate the sequential AVFs for every functional sequential in an Intel Xeon® core design, showing an average sequential AVF of 14% averaged over a variety of server workloads. This resulted in a ~10% reduction to the modeled overall processor SDC SER. Finally, we showed that the modeled results using the computed sequential AVFs improved our model correlation to silicon measurement by ~66% for a couple of workloads which brought the modeled SER well within the statistical range indicated by the measured values.

Bibliography/References

[1] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor", 36th Annual International Symposium on Microarchitecture (MICRO), December 2003

[2] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S.S. Mukherjee, R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures", 32nd Annual International Symposium on Computer Architecture (ISCA), 2005

[3] P. Hazucha, T. Karnik, S. Walstra, B. A. Bloechel, J. W. Tschanz, J. Maiz, K. Soumyanath, G. E. Dermer, S. Narendra, V. De, and S. Borkar, "Measurements and analysis of SER-tolerant latch in a 90-nm dual-Vt CMOS process", IEEE Journal of Solid-State Circuits, vol. 39, no. 9, pp. 1536–1543, September 2004

[4] S. Mitra, N. Seifert, M. Zhang, Q. Shi, K.S. Kim, "Robust system design with built-in soft-error resilience", Computer, Volume 38, Issue 2, Feb. 2005 Page(s):43 – 52

[5] N. Seifert, V. Ambrose, B. Gill, Q. Shi, R. Allmon, C. Recchia, S. Mukherjee, N. Nassif, J. Krause, J. Pickholtz, A. Balasubramanian , "On the Radiation-induced Soft Error Performance of Hardened Sequential Elements in Advanced Bulk CMOS Technologies", invited paper, proceedings of the IEEE International Reliability Physics Symposium (IRPS), pp. 188 - 197, 2010

[6] N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, V. Ambrose, Q. Shi, R. Allmon, A. Bramnik, "Soft Error Susceptibilities of 22nm Tri-Gate Devices", , IEEE Transactions on Nuclear Science, Volume 59, Issue 6, pp. 2666 - 2673, 2012

[7] 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, June 2004

[8] T. Dey, S. Raasch, J. Stephan, A. Biswas, "SDC Virus: An Application for SER Model Validation", Workshop on Silicon Errors in Logic - System Effects (SELSE), Stanford, California, April 2014

[9] http://en.wikipedia.org/wiki/Md5sum

[10] R. Nathan, D. Sorin. "Nostradamus: Low-Cost Hardware-Only Error Detection for Processor Cores." Design, Automation & Test in Europe (DATE), March 2014

[11] A. Avizienis. "Arithmetic error codes: Cost and effectiveness studies for application in digital system design." IEEE Trans. on Computers, C-20(II):1322–1331, 1971.

[12] H. Asadi, M. Tahoori, "Soft error modeling and protection for sequential elements," Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on , vol., no., pp.463,471, 3-5 Oct. 2005

[13] D. Holcomb, L. Wenchao, S. Seshia, "Design as you see FIT: System-level soft error analysis of sequential circuits," Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09. , vol., no., pp.785,790, 20-24 April 2009

[14] G. Saggese, A. Vetteth, Z. Kalbarczyk, R. Iyer, "Microprocessor sensitivity to failures: control vs. execution and combinational vs. sequential logic,"Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on , vol., no., pp.760,769, 28 June-1 July 2005

[15] J. Blome, S. Mahlke, D. Bradley, and K. Flautner, "A Microarchitectural Analysis of Soft Error Propagation in a Production-Level Embedded Microprocessor," Proceedings of the 1st Workshop on Architectural Reliability, 38th International Symposium on Microarchitecture, Barcelona, Spain, 2005

[16] X. Li, S. Adve, P. Bose, J. Rivers, "SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors", DSN 2005

[17] X. Li, S. Adve, P. Bose, J. Rivers, "Online Estimation of Architectural Vulnerability Factor for Soft Errors", ISCA 2008

[18] S. Hari, R. Venkatagiri, S. Adve, H. Naeimi, "GangES: Gang Error Simulation for Hardware Resiliency Evaluation", ISCA 2014

[19] H. Cho, S. Mirkhani, C. Cher, J. Abraham, S. Mitra, "Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design", DAC 2013

[20] A. Biswas, N. Soundararajan, S. Mukherjee, S. Gurumurthi, "Quantized AVF: A Means of Capturing Vulnerability Variations over Small Windows of Time", Proceedings of the 5th Workshop on Silicon Errors in Logic – System Effects (SELSE) 2009