Characterizing the Use of Program Vulnerability Factors for Studying Transient Fault Tolerance in Multi-core Architectures

Abstract

Semiconductor transient faults (soft errors) are a critical design concern in the reliability of computer systems. Most recent architecture research is focused on using performance models to provide Architecture Vulnerability Factor (AVF) estimates of processor reliability rather than deploying detailed fault injection into hardware RTL models. While AVF analysis provides support for investigating new fault tolerant architecture techniques, program execution characteristics are largely missing from determining periods of soft error susceptibility. The primary problem with AVF is that software periods of vulnerability substantially differ from micro-architecture periods of vulnerability. As research trends dictate finding ways to selectively enable software-based transient fault tolerant mechanisms, runtime and off-line experimental techniques must be guided equally by program behavior and hardware.

To address issues with AVF as well as the efficiency of fault injection studies, we examine elements of Program Vulnerability Factor (PVF) in the context of multi-core architectures. PVF has previously been introduced to consider program behavior in the form of memory/register vulnerability, however we explore static and profile based techniques for extending the work. By leveraging PVF we explore some initial contributions to the area of computer architecture research. First, we demonstrate that a more efficient fault injection campaign can be constructed and the outcome of fault injections in application execution can be accurately predicted. Second, compiler optimizations can be applied to better understand how the compiler affects fault susceptibility and program behavior. Finally, we motivate the need for developing a PVF metric for program data that is communicated between cores.

1 Introduction

With increasing transistor density enabled by the rapid scaling of CMOS technology and the rise of application complexity in recent years, next generation processors are being driven to integrate multiple cores per chip. The latest multi-core system-on-chip (SoC) architectures from Intel, AMD, Sun, Nvidia, Tilera, Xilinx and IBM replicate single processor cores many times on a chip die. Many designers envision future high-performance systems that integrate hundreds of multi-threaded cores per architecture, resulting in thousands of concurrent program threads sharing system resources. For these multi-core SoCs, CMOS scaling has been an enabler as well as a limiter: it has enabled unprecedented levels of integration, including the ability to integrate large memories with multiple processing cores on a chip, with the downside being a penalty in the power dissipation and reliability.

Next to power dissipation, transient faults are emerging as the next major concern for architects designing reliable computer systems [2, 12]. Trends in silicon process technology report a bleak future. While the future error rate of a single transistor is expected to stay relatively constant [6, 10], the number of transistors per chip continues to increase at an exponential rate. As a result, the overall error rate for processors is expected to increase dramatically, making fault tolerance as important a design characteristic as performance, power consumption, and temperature dissipation.

Hardware designs can be customized for fault tolerant execution with redundant resources such as latches or extended pipelines. Providing fault tolerance may require the addition of hundreds of thousands of delay latches and 20-30% logic to an existing processor [26]. Other more specialized approaches create even more sophisticated systems requiring both hardware and software integration [9, 33]. While these approaches work well in their specific scientific computing domain, the general purpose design field must adapt to the need for fault tolerance in fundamentally different ways. As design cycle time is critical, many chip designers propose implementing redundancy-based fault tolerance using existing multi-core and multi-threaded process extensions [5, 13]. The driving motivation is to extend the engineering decision towards multi-context processor to provide fault tolerance.

Most recent architecture research is focused on using
for(ptr=head; ptr->next!=null; ptr=ptr->next) 
{ 
    if (ptr->value == something) 
        sum += ptr->value; 
}

Figure 1. Linked-list code.

performance models to provide Architecture Vulnerability Factor (AVF) estimates of processor reliability rather than deploying detailed fault injection into hardware RTL models. AVF is defined as the probability that a fault in that particular structure will result in an error in the final output of a program [14]. A structure’s error rate is the product of the AVF, the number of bits, and the structure’s raw error rate, as determined by process and circuit technology. Processor designers can use AVF analysis to determine which processor structures are in probabilistic need of protection (e.g., structures with high AVF are likely to be protected). Some structures, such as the branch predictor, have no effect on whether an error will propagate to the output of the program. In contrast, other structures such as the instruction issue window, load-store queue, and re-order buffer have a resounding effect on program output. The majority of hardware structures fall in the middle of the two extremes.

While AVF analysis provides support for investigating new fault tolerant architecture techniques, program execution characteristics are largely missing from the determining periods of software error susceptibility. A software-centric view makes this key insight: although faults occur at the hardware level, the only faults which matter are the faults which affect software correctness. By changing the boundaries of output comparison to software, a software-centric model shifts the focus from ensuring correct hardware execution to ensuring correct software execution. As a result, only faults which affect correctness are detected. Benign faults are safely ignored. A software-centric system with both detection and recovery will not need to invoke the recovery mechanism for faults which do not affect correctness. The primary problem with AVF is that software periods of vulnerability substantially differ from micro-architecture periods of vulnerability. As research trends dictate, finding ways to selectively enable transient fault tolerant mechanisms, run-time and off-line experimental techniques must be guided equally by program behavior and hardware. As such, it is important to determine and predict when program susceptibility and hardware susceptibility differ.

To illustrate the differences between hardware and program vulnerability during execution, consider a synthetic example of a linked list in Figure 1. The AVF (including cache vulnerability as calculated in [1]) for the linked-list traversal will increase over execution time since most accesses of the list are cached in hardware. The AVF is illustrated in Figure 2. While some cache misses occur, the AVF will increase at a near linear rate until the cache is full. Once the cache is full, the AVF will stabilize. Minor changes in AVF value will vary slightly based on microarchitecture pipeline queues and buffers. Overall, since the linked list has no re-use of items already traversed, the AVF considers items not directly accessed (but having potential for tag matches differing by single bit error differences). While many recent works suggest that caches can be eliminated from consideration of transient fault vulnerability, the likelihood of constructing full cache hierarchy error correcting codes or parity remains unlikely and unobserved. In contrast, a program-level vulnerability factor for the linked-list would consider the potential amount of future memory state accessible. At the start of the traversal, a large program-based memory state exists, however, over time, fewer program-relevant memory elements could be impacted transient errors. Biswas calculated liveness via lifetime analysis and Architecturally Correct Execution (ACE) for tag-array and data-array components [3]. However, ACE analysis is still based on architecture structure (size/capacity). It is important to note that Sridharan [27] formally defined Program Vulnerability Factor (PVF) and evaluated the impact of input variation on PVF [28].

Figure 2 illustrates both the real AVF and PVF, as well as the profile versions of each metric. The two sets differ since the profile AVF will consider the elements in the cache to be dead. However, those bits are still resident and are suspect to single event upsets (SEU) at any given time. Meaning, a flip of a certain bit in the cache’s control bits, could adversely affect program output. Therefore, with each traversal the real AVF will increase, as the profiled AVF remains relatively constant.

There are several benefits to exploring software-based program analysis instead of only AVF during the design of fault tolerant processors. By only focusing on the architectural hardware structures, the community is limited in the full understanding of processor vulnerability within future multi-core architectures that involve both communication and computation. For these reasons, this paper investigates some extended aspects of program vulnerability. Moreover, since PVF is influenced by software characteristics, it offers better accuracy for understanding the impact that compiler optimization and code restructuring have on vulnerability. Experimental results indicate that PVF more accurately predicts actual fault outcomes compared with AVF. By leveraging PVF, an efficient fault injection campaign can be constructed to study the true nature of program execution and the design of transient fault tolerant techniques on modern processors.

The structure for the rest of the paper is as follows. Sec-
Figure 2. PVF and AVF of linked-list traversal.

2 Motivation

The software-centric approach to analyzing vulnerability provides a different model for identifying reliability issues, as well as providing fault tolerance within modern systems. For some computation domains, such as audio decoding and visual decompression [23], specific software has been shown to have tolerable error rates. From these observations, a number of software-based fault detection and tolerance techniques have been proposed. Executable assertions [7, 8] and other software detectors [17] explore the placement of assertions within software. Other schemes explicitly check control flow during execution [15, 22]. The pi bit [32] and dependence-based checking [30] have been explored as methods to follow the propagation of faults in an attempt to detect faults which affect program behavior. Software-based fault tolerance techniques [18, 19, 21, 25], in which redundant code is executed to ensure correctness, address many of these shortcomings but not in their current form.

The following sections investigate the fault tolerance of program behaviors in relation to fault injections. To model a transient error changing program state, the Intel binary instrumentation tool Pin [11] is used to modify a profiled architecture state bit.

2.1 Fault Vulnerability and Program Input Variation

The reliability of a modern processor varies based on a number of characteristics ultimately unrelated to the microarchitecture model. To examine differences in vulnerability, the PVF and the AVF for an x86 architecture (with parameters presented in Section 4) was collected for several sorting routines (radix, heapsort, merge, shell, quicksort) with three different (but same size) data inputs respectively - (r) random, (d) descending order, and (s) already sorted. In addition, each sort routine and input type were injected with 1000 faults.

Figure 3. AVF, PVF, and fault tolerance rates for radix, heapsort, merge, shell, and quicksort sorting routines.

The results in Figure 3, illustrates multiple sort routines, each having three different input data sets. Hence, one would expect each sort routine to have a varying AVF. However, as seen in the graph, architecture simulation-based AVF remains nearly constant for each input set, while the PVF and the fault tolerance rates show disparities while using the different inputs. These results suggest fault tolerance rates are highly dependent upon variations in function/program behavior and input. Program input changes fault susceptibility and this further illustrates why AVF cannot serve as the only evaluation mechanism when deciding which regions to protect against faults. A program vulnerability score PVF, which is introduced in the Section 3, closer correlates to the fault outcomes than AVF. The evaluation clearly demonstrates that when program behavior such as number of memory elements read before being written is taken into account, a more accurate view of vulnerability can be estimated.
2.2 Variability of Different Fault Injection Methods

One of the main problems the research community faces when evaluating a program’s fault tolerance level is that there is the lack of a good control model. In other words, no tool exists to accurately simulate or predict the entire fault tolerance rate of an application. Previous models randomly selected 1000 points in a program, flipped a bit in a register, and analyzed the affects of the program output [20]. This model has several weaknesses. First, 1000 random injections does not fully assess the fault susceptibility within a program. Without using any prior characterization of the running program, these random injections could occur at a program point which is complete fault tolerant or completely non-fault tolerant. More so, randomly placed errors will result in a non-deterministic analysis of the actual program’s actual fault tolerance rate. Therefore, the need for a better fault injection campaign is evident. However, it is impractical and virtually impossible to inject an error in every bit for every instruction in a program, thus the community needs a better and faster solution.

Using our Pin injection framework, we model several different injection methods. Once a fault occurs, the program is detached from Pin to continue its execution and to speed up simulation time. Results are determined by using the standard specdiff to analyze the error injected program’s output with the correct program output. Some injections cause abnormal behavior, e.g. a signal raised or a segmentation fault, and those runs are simply a subset of incorrect execution. Our fault injection variance analysis is done through the following faulting techniques:

- **Random** - Registers are faulted randomly throughout the entire execution. This approach is the worst at efficiently evaluating fault susceptible and fault tolerant regions.

- **Linear** - Uniformly injects faults throughout the entire application. This is done by finding the total dynamic instruction count of the program and injecting faults at uniform time slices.

- **Execution** - Frequently executed areas (e.g. functions or program counters) can be accountable for over thirty percent of a program’s execution. These areas are injected with errors to evaluate the fault tolerance rate of highly executed regions.

- **Phase** - Program phases are used to find which phase is executed the most and least, respectively. These phases are then injected at ten different time slices. More importantly though, the same instructions are injected within each injection slice further down the time line. Each phase is injected to see, if any, the different fault susceptibility of each phase.

The results in Figure 4 illustrates the differences in correct and incorrect execution of six SPEC2000 integer benchmarks and four SPEC2000 floating point benchmarks. The graph clearly indicates the disparity of results when using five different error injection methods described above (linear, execution, random, highest executed phase, and lowest executed phase). From these results, one can see the difficulty in creating accurate fault injection campaigns and illustrates two significant findings. First there is substantial variability across injection methods, differing upwards of 20% in some cases, as in 300.twolf. Each fault injection method results in different application fault tolerance rates. Second, the differences between injection results between the highest executing phase and the lowest executing phase can differ as high as 63% in the case of 176.gcc. These results indicate that fault injection results must be applied on more limited regions than whole program execution.
2.3 Impact of Program Phase Behavior in Fault Tolerance

The majority of applications have behaviors that are repetitive throughout different execution times. These execution intervals can be separated into phases using SimPoints [24]. SimPoints finds program phase behavior and is integral to architecture simulation, compiler optimization, and/or program analysis. PinPoints [16] is a Pin tool that derives the SimPoints phases, but uses dynamic instrumentation instead of simulation to find representative portions of programs. By using the phases of a program to direct fault injections, we hope to observe variation in fault outcomes that can be correlated to program characteristics. For this experiment, each phase is injected with a uniform distribution of faults. The results of four applications (164.gzip, 176.gcc, 253.perlbmk, and 255.vortex) are presented in Figure 5. The x-axis has the phases sorted from the highest execution weight to the lowest, with each phase labeled with its PinPoints number. The disparity between injection outcomes to higher and lower execution phases is easily discernible. Program phases with higher execution frequency have higher failure rates (e.g. 176.gcc - phases 3 and 5). Likewise, phases which execute less frequently have lower failure rates (e.g. 176.gcc - phases 4, 1, 0, 7, 2, and 6). These results are a source of motivation for inspecting whether the fault outcomes can be attributed to specific program structures or patterns, as well as whether the outcomes can be altered through program transformation, such as using compiler optimizations.

3 Approach

PVF encompasses more than traditional AVF and does not assume AVF accounts only for microarchitectural liveness. PVF is more of an enabler and should be used in conjunction with AVF to determine which program points need fault tolerant protection. Thus, PVF is not intended to fully replace AVF. By using PVF, designers and compiler writers can easily see the affects of program state. While AVF consists of finding the vulnerability of certain processor structures, PVF consists precisely of three main exclusive factors that determine a program’s execution pattern: Register, Memory, and Control Vulnerability.

There are three components of the PVF calculation. PVF is attributed to register, memory, and control vulnerability. Control vulnerability is based on a path score and dominator score, both indicating the likelihood of a branch error leading eventually to the correct path of execution. Register and memory vulnerability are based on reads of the data similar to reference [1]. Reads which are overwritten and which do not affect program outcome do not contribute to the overall PVF score.

PVF is different than AVF since PVF tracks program memory and not any simulated architecture components. Likewise, PVF defines program execution tracked to instruction points, not time (cycle) points. We analyzed PVF on the register starved x86 architecture, but PVF can easily be adapted to different architectures (e.g. PPC or IA64). By profiling, we were able to determine the correct weighting of each control, register, and memory component and evaluated PVF using a vulnerability cutoff of 10M instructions.

3.1 Memory and Register Vulnerability

PVF memory and register vulnerability is calculated by normalizing how long a register or memory reference remains open (before it is written over). There are several different references to memory and registers which cause different vulnerability scores. They can broken down into four groups: 1. Consecutive Reads, 2. Consecutive Writes, 3. Reads after Writes, and 4. Writes after Reads. The vulnerability for an instruction window is calculated by counting the number of instructions from the last previous write or read before another read occurs. When a write executes after either a previous read or write or when a read occurs when it will not be used again, the vulnerability is null. These reads and writes are commonly refereed as being first-level dynamically dead (FDD) [14]. The write is not used again and the value written does not impact the final program output. Thus, this score is not added to the overall score because it is meaningless. The length of time a register or memory location remains open directly corresponds to the probability of that register or memory location
to be prone to errors. The AVF calculation takes into account both direct and transitive register writes, eliminating dead operations from the pipeline. However, the direct or transitive nature of memory reads are only loosely adjusted in AVF. The mPVF value thus tracks a differences between long term memory use compared to cache vulnerability.

To calculate mPVF we keep track of reads, writes, and groupings of memory addresses. For each instruction window, each referenced memory address is grouped or hashed together by a mask. Then, the number of instructions (for a given memory grouping) between either a valid read or write are recorded. The final mPVF score is averaged by dividing the number of instructions by the instruction window size and then normalized by dividing the averaged result with the number of active memory bins - groupings that have a valid score. For our analysis, we found that by using an 8 bit mask on the address, we could achieve better coverage across different pages of memory. We also tested different instruction window sizes and found the most efficient size to be 10M. Calculating rPVF is very similar to mPVF. The main difference is that registers are not grouped together since there are a small number of register in the x86 instruction set. The averaging and normalization process remains the same.

3.2 Control Vulnerability

The concept of analyzing the affect of control execution on program state and fault susceptibility is an important because it augments the notion of program behavior. Control vulnerability factor (cPVF) is another major program attribute that impacts software vulnerability and is an element of the overall PVF. We model the calculation of our cPVF after the concept of Y-branches [31]. While the previous work examined changing the branch outcome (effectively a fault injection campaign) and studied approximately 1000 branch decisions, our analysis takes into account code behavior in the form of control flow graph properties. The cPVF score is calculated by checking for sets of compiler analyzed information for conditional branches. Each conditional branch is given a cPVF score, and the dynamic execution of branches are weighed to calculate the overall criticality presented by some branch operations.

The two compiler analyses used to construct the program cPVF are post-dominator sets and path selection sets. Post-dominator analysis indicates control blocks that are guaranteed to execute after a basic block. The path selection set consists of the set of reachable basic blocks given a fixed number of branch executions. In this analysis, the fall-thru and taken blocks of every conditional branch are analyzed. The goal of using these derived sets are that they should reflect critical branches; those conditional branches that will lead to substantially different blocks of the program if a wrong decision is caused by an earlier transient fault. Figure 6 demonstrates two example cPVF calculations. First, is the example of a basic if-then-else branch. In this case, the conditional branch could take the wrong path since the post dominator block as well as the path of potential blocks are identical. However, a second case is presented in Figure 6 with a more complex control graph occurring after the conditional branch at block BB1. In this case, both the post-dominator set and path set intersection are different by a few degrees compared to the individual post-dominator and path set intersection of BB2 and BB3. The cPVF score for each conditional branch is a fraction derived by the average of the intersection size of PDOM with the largest target’s PDOM set, and the intersection size of the PATHSET over the maximum target block’s PATHSET size. A path selection depth of 10 branches was used for evaluation (which compares the blocks reachable by 10 branch executions from both sides of the conditional branch). The cPVF method is calculated in an off-line way as to not require run-time analysis of each branch direction/outcome. Future work will correlate the change of cPVF scores with specific compiler transformations.

Figure 7 reports the breakdown of static conditional branches in the SPEC2000 integer benchmark applications. Each application is rated as not critical, medium, high medium, and critical. The cPVF critical operations are those conditional branches having high PDOM and PATHSET scores. The cPVF metric provides insight into the benchmark vulnerability by estimating the existence and occurrence of conditional branches leading to dramatically different code if wrongly executed. Benchmark 252.eon is interesting from the perspective of having very few critical conditional branches. Otherwise, 186.crafty and 300.twolf have the highest percentage of critical branches.
4 Results

In this section, results from the PVF and AVF experiments are showcased. Specifically, time and phase-varying behavior is noticed in every benchmark. These results support the notion of how important it is to decide whether or not to turn on fault protection. There are significant trends and correlations between AVF, PVF, and a program’s output (when injected with errors).

4.1 AVF and PVF Experimental Setup

All of the AVF experiments were conducted using a modified version of the SimplerScalar 3.0 simulator [4]. The standard version supports Alpha and PISA binaries, while the version used for our calculations support x86 binaries. Due to the nature of the simulator, all binaries were compiled using the -static flag. Modifications were made to the standard simulator in order to compute the AVFs of Reorder Buffer and Physical Register File (RUU), the Load/Store Queue (LSQ), and the Issue Queue.

The AVF evaluations were run on all 26 integer and floating point benchmarks of the SPEC2000 suite [29]. The configuration of the processor used for this analysis is given in Table 4. PVF was modeled using the Intel Pin binary instrumentation tool [11]. The evaluation of fault propagation and performance were both analyzed using the reference input set.

4.2 PVF-AVF Comparison

The following section illustrates how PVF can differ from AVF and why there needs to be another metric for hardware designers to use in order to efficiently use fault tolerant protection mechanisms. Figure 8 illustrates how AVF and PVF differ over time. Out of the three benchmarks only 176.gcc stays relatively consistent. The other two benchmarks (186.crafty and 300.twolf) display numerous positive spikes which indicate AVF is varying while PVF is remaining constant. Using PVF, these spikes can be predicted and traced back to program code points. Further analysis will allow for designers to enable certain fault protective regions of code.

4.3 Predicting the Error Injection Rate

There are no accurate fault tolerant prediction tools in existence. While comparing PVF and AVF, we found that for certain regions of execution, the PVF score better mirrored the percentage of correct and incorrect injections, than the AVF score. Figure 9 illustrates how different AVF and PVF are in the prediction of each integer SPEC2000 benchmark. In the graph each benchmark has two bars, the left represents AVF and the right represents PVF. There were 1000 experiments and the y-axis indicates the percent of the experiments in which the benchmark’s AVF/PVF score is within the benchmark’s fault tolerance rate. For example, for 175.vpr the PVF score is within a 0-10% range of the fault tolerance rate, for 80% of the experiments and within a
10-20% range, for only 20% of the experiments. In contrast, for 175.vpr, the AVF score is within 0-10% and 10-20% of the fault tolerance rate, for 20% of the experiments each and within 20-30% for 60% of the experiments. While AVF resembled the fault rate for a couple of the benchmarks, PVF predicted the correct fault tolerance output within a 10% tolerance of 25% of the runs. These results indicate that PVF can be used as a better metric than AVF to predict the fault tolerance rate of an application. This is because PVF models program behavior and not the micro-architectural state. Hence, program execution patterns can be matched to the fault tolerance rate within a reasonable percentage.

### 4.4 Compilation Induced Variance on AVF and PVF

The compiler plays a critical role in determining program execution characteristics. To investigate the effects of compiler optimizations on application fault tolerance, we constructed an experiment based on the GNU Compiler Collection (GCC)’s optimizations. Using Gcc4.1, we analyzed 40 optimization flags. Our experimental system randomly selected 8 compilation vectors each composed of the 40 optimizations turned on or off. By varying compilation flags and optimizations a compiler can affect an application’s overall AVF, PVF and fault susceptibility. The experiment was ran on the 12 SPEC2000 integer benchmarks and results of the 8 distinct compilations are shown in Figure 10 and Figure 11. Figure 10 illustrates that when AVF and PVF are evaluated under eight different compiler variations, AVF and PVF contrast each other. Out of the 12 benchmarks, four of them (186.crafty, 254.gap, 255.vortex, and 256.bzip2) had a high variation of AVF, while PVF only varies significantly in only one benchmark. The standard deviation of all the compilations averaged across each benchmark was 2.3% and 0.7% for AVF and PVF respectively. The standard deviation change suggests that PVF may be a more stable metric for anticipating the fault tolerance of a program’s execution.

There are several trends that can be observed in Figure 11 regarding the compiler’s ability to impact both AVF and PVF vulnerability metrics. First, most applications have stable behavior, with AVF having higher variation. However, both AVF and PVF scores vary greatly across compilations for application 176.gcc, which is a large application with complex control and data access patterns. Second, certain compilations (5) significantly impact the AVF score without altering the PVF score. This result indicates that there are ways to transform the micro-architecture efficiency that do not change the inherent program state. Further analysis of how compiler optimizations precisely change program behavior is needed.

### 4.5 Analysis of Communication Vulnerability Factors in Multicore Architectures

Multicore SoC architectures can encounter transient, intermittent, and permanent faults that can have a detrimental influence on various aspects of system behavior. Figure 12 shows an example of a multicore SoC with 16 cores, with a 2D packet switched Network on chip (NoC) communication fabric. The NoC fabric consists of network interfaces (NIs) that computation cores use to inject or receive data from the communication network. The NIs typically split data packets into smaller sized flits, and send the flits to a local router (also referred to as switch) which is connected
to neighboring routers using interconnection links. Each of these routers typically uses a routing scheme to determine the shortest path to the destination for the flits. Once the flits arrive at the destination NI, they are combined and forwarded to the destination core.

In the presence of faults, this normal behavior of the system can be easily disrupted. Consider the scenario from Figure 12 where data must be sent from core A to core B. In the absence of faults, the packet would traverse in a straight line, vertically, traversing four routers. However, if a router in the path has a permanent fault, the packet needs to be routed along an alternate path. Permanent faults can occur on links as well. Thus the shortest path for the packet in the presence of the link and router failures shown is to go west from router 1 to 4, then north from router 2 to 3, and then east from router 3 to 4.

The need for on-chip fault-tolerant communication started well before the NoC paradigm emerged as a solution for integrating large multicore SoCs. Solutions for fault-tolerant on-chip buses have used duplication to detect and correct crosstalk and transient faults, and where different error recovery mechanisms from simple retransmission to correction/ retransmission are analyzed in terms of power/area. NoC fabrics comprise of more complex topologies with higher degrees of connectivity, compared to buses. However, the additional hardware and information redundancy in NoCs offers significant potential for implementing different fault-tolerant strategies. Traditionally, communication fault tolerance schemes have proposed adaptation at two levels: data link level, and network level.

As part of our future research plan in this area, we will evaluate the vulnerability of communication content and communication patterns. We plan to have these results evaluated by the time of the workshop presentation and publication. We believe from this exploration will show opportunities for adaptively routing packets from the source to destination will be successful in adjusting to the presence of faults in NoC links and routers.

5 Conclusion

This paper investigates the program factors which contribute to the execution vulnerability in modern processors. Overall, our goal was to provide a metric that does not depend on simulation inaccuracies. For example, AVF has not been calculated with OS context switching of program state or multi-threaded interactions of multi-core shared caches damaging the cache. As such AVF simulation techniques need to be extended to program behavior. AVF has been integrated into the design exploration of new architectures and new fault tolerant techniques. However, AVF is limited in the sense it only deals with hardware and ignores
program execution characteristics. To this end, we propose PVF, a method of evaluating a program’s vulnerable execution state. While PVF varies for different applications, over different execution times, and different phases of a program, it offers an accurate way for predicting vulnerability behavior. PVF improves upon AVF as it can easily be used to characterize program execution behaviors, construct a more efficient fault injection campaign, and explore effective compiler optimizations that may increase a program’s fault tolerance rate. Experimental results show that compiler optimizations have a greater impact on the AVF score, while other results indicate PVF is better able to track fault tolerance rates.

6 Future Work

Future work will leverage PVF to guide both static program analysis and run-time fault tolerance techniques. For static analysis, we wish to develop a technique to provide checkpoints for general program regions to determine perturbed program execution. Programmers will be able to use PVF to find program spots which are in critical need of fault tolerant protection. Likewise, PVF profiling can guide software-directed run-time fault tolerant techniques by indicating critical regions to protect with redundant techniques. Thus, improving power, temperature, and performance metrics. We have ideas for extending PVF analysis to multi-threaded applications (both tracking shared memory state/values and co-phase behavior). The goal of the co-phase behavior analysis would be to determine critical/likely state for program errors.

References


