

# Perturbation-based Fault Screening

Paul Racunas<sup>1</sup>, Kypros Constantinides<sup>2</sup>, Srilatha Manne<sup>3</sup>, and Shubhendu S. Mukherjee<sup>1</sup>

<sup>1</sup> FACT Group, Intel Corp.  
Hudson, MA 01749

<sup>2</sup> Dept. of Computer Science  
and Engineering,  
University of Michigan,  
Ann Arbor, MI 48105

<sup>3</sup> ITPP Group, Intel Corp.  
DuPont, WA 98327

## Abstract

*Fault screeners are a new breed of fault identification technique that can probabilistically detect if a transient fault has affected the state of a processor. We demonstrate that fault screeners function because of two key characteristics. First, we show that much of the intermediate data generated by a program inherently falls within certain consistent bounds. Second, we observe that these bounds are often violated by the introduction of a fault. Thus, fault screeners can identify faults by directly watching for any data inconsistencies arising in an application's behavior.*

*We present an idealized algorithm capable of identifying over 85% of injected faults on the SpecInt suite and over 75% overall. Further, in a realistic implementation on a simulated Pentium-III-like processor, about half of the errors due to injected faults are identified while still in speculative state. Errors detected this early can be eliminated by a pipeline flush. In this paper, we present several hardware-based versions of this screening algorithm and show that flushing the pipeline every time the hardware screener triggers reduces overall performance by less than 1%*

## 1. Introduction

Radiation-induced soft errors—caused by neutrons in cosmic rays or alpha particles in packaging material—are becoming an increasing burden for microprocessor designers. The raw error rate per device (e.g., latch, SRAM cell) in a bulk CMOS process is projected to remain roughly constant or decrease slightly for the next several technology generations [10][11]. Thus, unless we add more extensive error protection mechanisms or use a more robust technology (such as SOI), a processor's error rate will grow with Moore's Law in direct proportion to the number of devices we add to a processor in each succeeding generation.

Industry limits the soft-error exposure of their processors in three ways: by adjusting materials, by designing redundant circuits and by architectural techniques. Every silicon manufacturer makes trade-offs between cost, performance and error protection. Error protection mechanisms, such as radiation-hardened circuits or architectural redundancy can come with significant penalty in performance,

power, and area. Hence, excessive protection may make the resulting product uncompetitive in cost and/or performance. Alternatively, a microprocessor with inadequate protection from soft errors may prove useless due to its unreliability.

This paper is targeted at architectural solutions to faults. The two traditional architecture-level solutions to processor fault tolerance are use of processor lockstep and addition of error-detection circuitry (parity & ECC) to processor components. Lockstep allows large portions of the chip to be covered with a single unified hardware mechanism at the cost of halving the throughput of the system. (Recently, techniques such as Redundant Multithreading [14] have been introduced to provide the logical benefits of lockstep with increased throughput.) Parity and ECC allow an individual single on-chip structure to be protected, often without a significant performance impact. It would be difficult, however, to protect every processor structure and logic with ECC without incurring significant penalties in area, power, or performance.

The contribution of unprotected structures to a processors soft-error rate can be reduced by probabilistic fault tolerance mechanisms. We use the term *fault screener* to refer to a number of probabilistic fault tolerance solutions that exist in the space between lockstepping and full ECC solutions. A fault screener operates by examining program state for internal inconsistencies with past behavior. We call a departure from established program behavior a *perturbation*. Consider a static instruction that generates a result value between 0 and 16 the first thousand times it is executed, then generates a value of 50. Since the new value of 50 does not match the pattern of established behavior for that static instruction, the new value is an example of a perturbation. When a perturbation is discovered, a fault screener will speculatively trigger a recovery, for example, by flushing the processor pipeline. If the perturbation was due to correct program execution, it will repeat when processing resumes. If the perturbation was due to a soft error, it will likely be eliminated by the pipeline flush. Fault screeners provide a method of reducing the soft error contribution of generic logic and processor structures that cannot be protected by ECC due to cycle time or area issues. This reduction may allow a processor to satisfy its soft error

rate requirements without explicitly incorporating full protection.

This paper makes four contributions.

- We show that fault screening is an effective technique for bounding application behavior and identifying suspicious changes in that behavior by looking for program perturbations. We show that an idealized screening algorithm can catch > 85% of faults on the SpecInt suite and over 75% of faults overall. This level of fault detection could potentially allow a design to meet its SER goals without resorting to extreme measures. We show that, on average over the SpecInt & SpecFP benchmarks, using the fault screening algorithms described, the incidence of detectable perturbations increases by more than a factor of 30 in the presence of a fault compared to the level of perturbations in the same set of dynamic code segments in the absence of a fault.

- We analyze the performance of fault screeners based on value history, dynamic range, bit-invariance, and a bloom-filter and show the strengths and weaknesses of each. We compare these results against other implementations of fault screeners in related work, both from the hardware and software communities.

- We monitor the propagation of architectural state faults in detail both in the short term via a microarchitectural simulator, and in the long term via a binary modification tool. This allows us to examine both the immediate effects of faulty data on dependent surrounding instructions and the long-term user-visible effects that the faulty data has, such as causing a segfault or changing the output of the program.

- We present a sample hardware implementation of the invariance-based fault screener and show that nearly half of the faults that would have eventually caused an output error or early program termination can be caught and repaired while still in processor speculative state. The performance loss due to flushing the pipeline on all false positives is less than 1% for this implementation.

The rest of the paper is organized as follows. Section 2 describes some related work. Section 3 describes the theory behind perturbation detection and the general operation of fault screeners. Section 4 describes the fault screeners we will be analyzing in the paper. Section 5 describes the three simulation frameworks used to generate results for the paper. Section 6 presents the results analyzing each fault screener. Section 7 presents a sample hardware implementation of a fault screening algorithm and Section 8 summarizes and presents our conclusions.

## 2. Related Work

Network intrusion detection systems [2][4][7][16] have long been using anomaly-based detection to find known intrusion signatures or anomalous network behavior in the networks that they monitor. Anderson [2] presented the

notion of intrusion detection using surveillance to collect data on the behaviors of individual users that could be collated to look for suspicious activity. Ghosh [7] used system call traces of running applications to characterize the typical behavior of the monitored system. Subsequently these reference traces were used to build, either automatically or with manual assist, finite automata that were capable of monitoring the system and reporting any deviations from the characterized normal behavior.

Eraser [17] used anomaly-based detection to create a system that monitors the locks protecting variables in multi-threaded programs and triggers a warning when inconsistencies in the usage of locks are detected. Hangal and Lam [8] showed that anomaly-based detection can be used to identify software bugs in Java programs. They did this by forming dynamic hypothesis about program invariants, and checking these invariants at runtime. An unexpected change in a program invariant was reported as a possible software bug. Research into value prediction [9] has shown that static instructions have considerable value locality. Because this locality exists, fault screeners are able to treat departures from established locality or patterns (perturbations) as anomalous events and speculatively trigger a recovery.

There are two components to a fault screener: a screening algorithm and a fault elimination mechanism (often a pipeline flush or checkpoint recovery). A history of hardware fault screeners already exists in the literature. Weaver, et al. [20] presented a scheme to flush certain pipeline structures on infrequent long latency events, such as cache misses. This allowed them to eliminate errors attributable to having stalled values exposed to soft errors for long periods of time while the cache miss was being serviced. The performance loss due to flushing was minimized since the processor was already stalled. Their screening algorithm was designed to identify stall situations where a flush would not prohibitively hurt performance, but it did not directly try to identify signs that a fault had actually occurred. Armstrong [1] presented a screening scheme (not related to transient faults) designed to improve performance by resolving branch mispredictions early. Their screener triggered a fetch redirect on certain suspicious events such as null pointer dereferences and arithmetic exceptions. They showed that these events were more likely to occur on an incorrect control flow path, and that their occurrence often served as an early indication that a previous branch had been mispredicted. However, many mispredicted branches did not generate null dereferences or exceptions. Lastly, Wang [19] showed that transient faults may directly cause the misprediction of a biased branch or cause a spurious cache or TLB miss. By flushing the processor pipeline on these events, they were able to significantly reduce the vulnerability of their machine at a smaller performance penalty

than a redundant solution. However, their fault detection latency was high, meaning that explicit checkpoints would be required, and their false positive rate was large enough to significantly affect performance.

### 3. Perturbation-based Fault Screening

A *fault screener* is a probabilistic fault tolerance mechanism that uses historical information or current processor state to establish a profile of expected behavior, and that signals a warning when encountering behavior outside of that profile. A fault screener operates by monitoring *indicators*. An *indicator* is any event that does not often occur in normal program operation, but that may occur in the presence of an event of interest. In the network intrusion detection world, an indicator may be a system call trace that was previously seen as part of a network attack. Hardware-based fault screeners have used anything from arithmetic exceptions [1] to TLB misses [19] as indicators that something may be wrong with program state. Naively, an indicator is any event that suggests that the program has departed from its established behavior.

We call a departure from established behavior a *perturbation*. A perturbation may be *natural*, resulting from variations in program input or current phase of the application, or may be *induced*, resulting from a fault. Consider a static instruction in an algorithm that happens to generate a result value between 0 and 16 the first thousand times it is executed. Its execution history suggests that the next value it generates will also be within this range. If the next instance of the instruction instead generates a value of 50, this value is a deviation from established behavior and is considered a perturbation. The new value of 50 could be a natural perturbation resulting from the program having moved to process a new set of data, or it could be an induced perturbation resulting from a fault. Fault screeners can often not distinguish natural perturbations from induced ones, and therefore require an efficient recovery mechanism capable of managing the false positives they generate. Fault screeners are also incapable of catching faults that do not cause program perturbations, unless the fault coincides with an unrelated natural perturbation. However, if coverage rates are high enough, fault screeners may provide multi-component coverage at a low performance overhead. This allows the end-user more options to dial-in his required level of reliability.

While fault screeners can be wrong, for example by calling an event a perturbation when the event is consistent with past behavior, they do not make predictions. We define a prediction to mean a forecasting of a future event. Since the fault must have already happened to cause a perturbation in program state, the screener is not predicting that a fault has occurred, but using post-calculation analysis to identify that it has occurred. Post-calculation analysis has

tighter latency requirements than predictions as the perturbation resulting from the fault must be identified before the original fault is written to non-recoverable state. While we target our techniques at identifying faults before the faulty instruction is retired, a checkpointing mechanism could be employed to allow a fault screener additional time to identify a program perturbation.

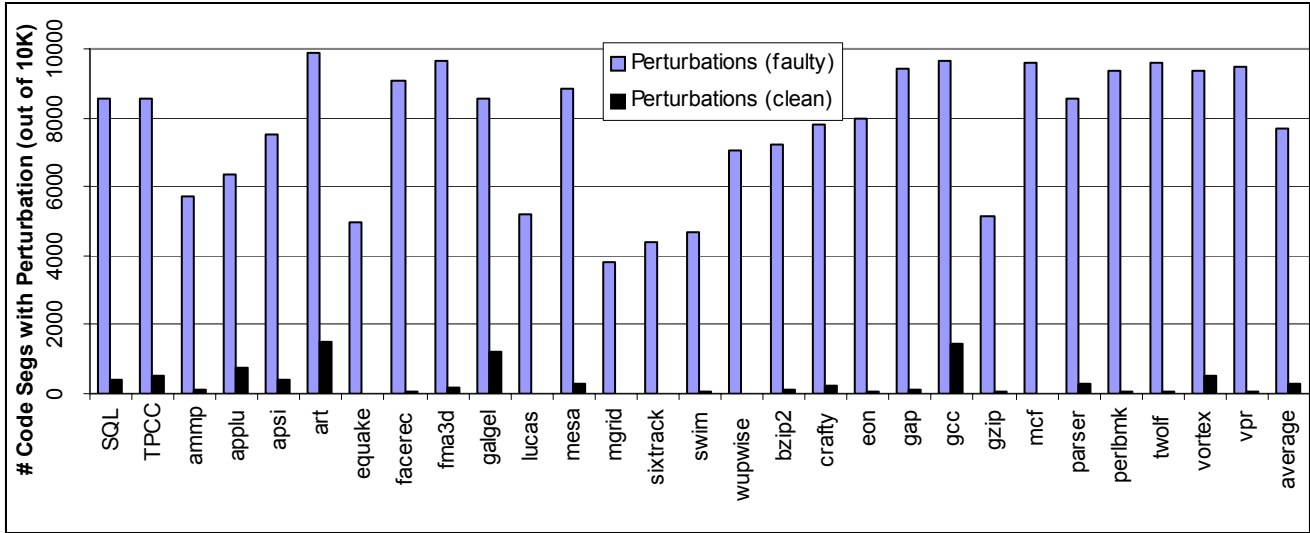
#### 3.1. Perturbation Definition

We have defined a perturbation as a departure from established behavior. This definition encompasses a large set of possibilities for establishing behavior, from static relationships identified by a programmer such as program asserts, to information collected dynamically during the program's run such as expected data ranges, to inconsistencies in current microarchitectural state, such as the valid bits of a queue disagreeing with the position of the head and tail pointer. However, for the purpose of this paper, we will attempt to establish program behavior patterns only dynamically during program execution. Also, the only dynamic analysis we will perform is to attempt to tie a static instruction or group of instructions to its expected valid value space. We do not claim that this approach is optimal, as one may be able to establish tighter bounds on the valid value space by monitoring other information, such as the interaction between static instructions. However, we find that using this approximation allows us to observe a good fraction of fault-induced perturbations.

A static instruction's valid value space is the set of result data values that could be written by the next dynamic instance of that instruction without being inconsistent with the current application state. For the purposes of this paper, we make the determination of valid value space by looking at the static instruction's past result behavior and extrapolating expected future behavior. We use standard techniques to extrapolate the possible future value space, such as relying on temporal and spatial value locality and trying to establish a set of possible strides for the instruction.

In order to implement a perturbation detection mechanism, we need to create a specific operational definition for a perturbation. From a practical standpoint, a perturbation is easiest to identify when associated with a static instruction with a great deal of value locality or a regular access pattern, and harder to identify when the associated static instruction generates many varied values. Hence, we classify static instructions by their result value locality and use different criteria to define a perturbation for each class. We separate the set of static instructions into three classes.

The first class consists of static instructions whose working set of results is less than 256 unique values. We populate this class by identifying static instructions that generate result values hitting in an array of 256 unique values 99.9% of the time. For static instructions falling into



**Figure 1. Effects of fault on perturbation rates**

this class, a perturbation is defined as any new result that cannot be found in this array of 256 unique values

The second class consists of static instructions whose result values have an identifiable stride between consecutive instances of the instruction. We maintain an array of 256 most recent unique strides for each static instruction. Each stride is computed by subtracting the most recent result value generated by the instruction from its current result value. This class is populated by the set of static instructions whose strides can be found in this array at least 99.9% of the time. For static instructions falling into this class, a perturbation is defined as any new result that produces a stride that cannot be found in the array of 256 unique strides.

The third class consists of any static instruction not falling into either of the first two classes. For these instructions, the first result value they generate is recorded. Each time a new result value is generated, it is compared to this first result, and any bits in the new result that differ from the first result are identified. A bitmask is maintained that represents the set of all bits that have never changed from the first result value through the entire course of program execution. For this class of static instruction, a perturbation is defined as any new result value that differs from the first result value in a bit location that has previously been invariant. After each perturbation occurs, the bitmask is changed to mark the new bit as variant. Hence, a static instruction with a 32-bit result can be responsible for a maximum of 32 perturbations in the course of the entire benchmark run. The method of detecting invariant bits in this class is similar to the method used in [8].

### 3.2. Fault Effects on Perturbations

Figure 1 shows how the number of program perturbations as defined by the three classes above increases in the presence of faults. For this graph, we run each benchmark ten thousand times. On each program run, we pick a random segment of dynamic execution 64 instructions in length. The black bar in the graph represents the number of these 10,000 segments that contain a program perturbation during the normal run of the benchmark. These represent natural perturbations. For the benchmark SQL, we see that 386 of these 10,000 random segments contain a natural perturbation. We then redo the benchmark runs, this time injecting a fault into the program just before each of the same random segments of execution. The gray bar represents the number of these segments that contain a perturbation after the fault is injected. The difference in magnitude between the gray bar and the black bar represent the segments that contain fault-induced perturbations. The benchmark with the most number of segments containing natural perturbations is art, which contains natural perturbations in 1,524 of the 10,000 program segments. When the same dynamic program segments of art are examined 64 instructions after a fault injection, there are perturbations in nearly all of the 10,000 program segments.

The graph shows that the number of instances of program perturbations increases more than 30-fold on average in the presence of a fault. This is the fundamental phenomenon that our fault screening mechanisms rely on for fault identification.

On average, over 75% of injected faults can be identified within 64 instructions of the injection point by using this algorithm. For the Spec integer benchmarks, on average over 85% of the injected faults can be identified because they cause perturbations as we have defined them.

Since our fault screener algorithms attempt to identify program perturbations, this graph represents a bound on the overall coverage possible through the use of the techniques presented in this paper. The graph shows that in several of the floating point benchmarks (mgrid, sixtrack, swim) the majority of the injected faults do not cause program perturbations, and hence are effectively undetectable with our current methods. This is because these programs have a set of static instructions that each produce a large set of result values. There are few bits of these result values that remain invariant during the course of the program's execution. It is possible that the value space of these instructions could be better represented if we treated floating point results as the value they represent in IEEE format, rather than as a set of discrete bits. A better value space representation may allow us to identify more perturbations than are identifiable with our current methods.

### 3.3. Fault Screener Evaluation Metrics

There are three important metrics to observe in the evaluation of the performance of a fault screener. First, fault coverage, meaning the number of injected faults that were successfully identified, should be as high as possible. Second, the accuracy of the screener should be high, meaning that in the course of normal valid execution only a small number of false positives should be generated. If our fault screener were able to identify every perturbation in Figure 1, each natural perturbation would represent a false positive. Accuracy is important as it is simple to have high coverage by assuming that every instruction is faulty, but performance will suffer considerably because the pipeline is flushed on every detected perturbation. Third, the latency of detection is also important. Since fault screeners perform post-execute analysis, the identification of a fault's perturbation cannot start until after the original faulting instruction is executed and must complete before that instruction is committed. A checkpointing mechanism could be included to extend this time period, but checkpoints cannot be easily taken across I/O boundaries and would require hardware or software to record lists of the memory data that differs in the checkpoint.

## 4. Description of Fault Screeners

In this section, we will present a list of the fault screeners that we analyzed, and describe the advantages and disadvantages of each. The screeners can be distinguished almost entirely by their varying representations of what should be considered the valid value space of the instructions of a program. The valid value space represents the set of values that would not generate a program perturbation if generated by its associated static instruction.

### 4.1. Valid Value Space Representation

While we were able to represent the valid value space fairly exhaustively in the perturbation definition of Section 3.1, a realistic fault screener has limits to the amount of information it can store. Once the possible future value space has been identified, it must be represented in some reasonable form. Doing this often requires generating an approximate representation of the value space. While it is possible to exhaustively record every result value generated by a program, it is cumbersome to represent a static instruction's expected future value space. There are many ways to represent even simple future plausible data behavior, such as allowing for value locality and fixed strides. Moreover, since program data will have temporal locality, a result that makes sense during the initialization phase of a program may be unreasonable during the program's final output phase. Exhaustively recording result values will likely not result in an efficient representation of a static instruction's current valid value space. Because of these factors, we present data for several different representations of a program's valid value space, and discuss the relative advantages of each. In fact, representation of valid value space is the main distinguishing factor between all of the fault screeners presented in this paper.

It should be noted however, that the problem of representing an instruction's valid value space is much easier than the problem of predicting the instruction's future values. In our case, we need not predict what the result of an instruction will be next, we only need determine what it should *not* be.

### 4.2. Extended History Screener

The extended history screener is based on the perturbation identification algorithm described in Section 3.1. A history of 64 unique values is kept for each static instruction. In addition, a history of 64 unique deltas between successive values is also kept. If a new value written by the static instruction is not in the history of 64 unique values and also does not match any of the 64 last unique deltas from successive values, the new value is considered to be a program perturbation. To manage static instructions that generate so many unique values that they cannot be bounded with this methodology, any static instruction that generates more than one perturbation for every hundred dynamic instances is filtered. Filtered instructions cannot cause a program perturbation no matter what result they generate. Filtering is included to keep the false positive rate close to 1 false positive per 1000 instructions. The advantage of the extended history screener is that because it lists individual values, it can detect small perturbations that other screeners may miss. The disadvantage is that only a small valid value space can be represented, and perturba-

tions generated by static instructions with large valid value spaces will not be identified.

### 4.3. Dynamic Range Screener

This screener dynamically builds a range representation of the valid value space by dividing it into resizable segments. A screener implementation has a set number of range points, where a range point is a segment of the value space represented by a maximum and a minimum value. Initially a range point is assigned to each new value as it is encountered. When all the range points are full and a new value is encountered, the range points are arranged around the new value so that all previous values are encompassed with the minimum amount of valid value space possible. The basic operation of the dynamic range screener can be seen in the example below:

**Table 1: Operation of dynamic range screener**

New Result Value	Range Point 1	Range Point 2
5	inval	inval
72	5-5	inval
6	5-5	72-72
5004	5-6	72-72
	5-72	5004-5004

Table 1 shows the operation of the dynamic range screener with two range points for a given set of input values. Initially, both range points are invalid. When the instruction first writes the result 5 and then the result 72, each of these values gets their own range point. When the value 6 comes along, it is merged into the first range point, as this covers all encountered values with the minimum of specified value space. Lastly the value 5004 comes along. Since 5004 is so much larger than 5 and 72, it is given range point 2 to itself, and the 72 from range point 2 is merged with the data in range point 1. At the end of the example, the dynamic range screener will trigger on any value  $X$  if  $((X < 5 \text{ or } X > 72) \text{ and } X \neq 5004)$ . Any values between 5 and 72 will be accepted as valid, as will the value 5004.

The base dynamic range screener generates many false positives in the presence of stride-based operations, because each new unique value will trigger the screener and force a range point modification. To reduce the false positives due to this effect, we calculate a stride based on the last value that the static instruction generated, and update the range points with both the current value and the current value + stride. Therefore, each access to the dynamic range table actually results in two updates. The dynamic range screener also employs a filter similar to the extended history predictor to keep its false positive rate down. The advantage of the dynamic range screener is that it can represent a large valid value space. The disadvantage is that it

can be too quick to declare large swaths of the value space to be valid when range points are merged.

### 4.4. Invariance-based Screener

This fault screener is adapted from the Diduce software anomaly detector by Hangel and Lam [8]. Their anomaly detector was used to detect software bugs in Java code. It operated by associating a bitmask with certain critical instructions in the program. The bitmask would indicate, for each critical static instruction, which bits of the result were invariant through the course of the program’s execution. Any change to these invariant bits could result in a warning issued to the user.

Our adaptation maintains a bitmask associated with each static instruction in the program. The last value written by that static instruction is recorded. As each subsequent value is encountered, the bitmask is updated to represent which bits of the result are continually invariant across all dynamic executions of that static instruction. A delta invariance is also performed. The delta is obtained by subtracting the recorded last result value from the current result value. A delta bitmask is then adjusted to indicate what bits of this delta have varied during the program’s run. A warning is flagged when a new bit is set in either the value-based bitmask or the delta bitmask. The invariance-based screener does not require a filter to maintain a low false positive rate. The advantage of the invariance screener is its efficient value space representation and low false positive rate. The disadvantage is that it is particularly affected by destructive aliasing.

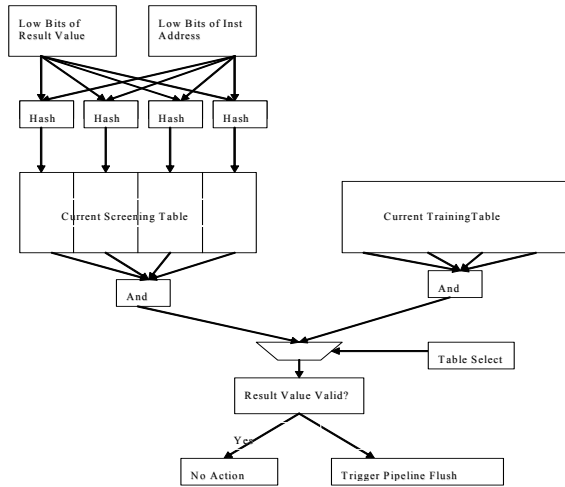
### 4.5. TLB-based Screener

The TLB-predictor is derived from a fault screener suggested by Wang [19]. Any memory access causing a TLB miss is treated as a program perturbation. Any memory access not causing a TLB miss is accepted as normal. A 4K page size is used for the TLB. The main advantage of the TLB-based screener is that it comes for free, using an already existing hardware structure to provide its screening ability. This fault screener is best at identifying errors that propagate to the high-order bits of data addresses. It is incapable of identifying perturbations that do not affect the address stream of the program. It is also incapable of identifying perturbations that only alter the low-order 12 bits of the data address. Its main disadvantage is its high false positive rate and comparatively long detection latency.

### 4.6. Bloom Filter Screener

This fault screener was included to show that a large amount of state is not necessarily required to represent a program’s valid value space. This screener is based on Bloom filters [3], and its operation is detailed in Figure 2. The central concept is to hash the instruction result and the instruction address together several separate ways, and

index into a table with each of the hashes. If any of the hashes returns a bit that is not set, the current result address and instruction address represent a pairing that has not been seen before, and a pipeline flush is triggered. After indexing the table to determine the value's validity, each of the accessed bits in the table is set to 1, meaning that this instruction address and data value pairing will be allowed in the future. To prevent all the tables from saturating with 1's, the table is reset regularly. To limit the number of false positives resulting from resetting all bits in a table to 0, two separate tables, indexed identically, are used. Both tables are updated continuously, but only one is used to generate a screening test. At regular intervals, the table not currently being used for screening is reset and screening tests are switched to come from the other table. This mechanism allows both tables to remain up-to-date, while also permitting combinations of data values and instruction addresses that have not been seen in a long time to once again be treated as perturbations. The bloom filter used in our experiments is 1Kbytes in total size.



**Figure 2. Diagram of Bloom Filter Screener Operation**

## 5. Methodology

This section will describe each of the three different simulation frameworks that we used within this paper. Section 5.1 discusses how we inject faults. Section 5.2 describes the ASIM functional simulator, which has been augmented with a high-bandwidth fault injection engine useful for focusing on short-term propagation effects of faults. The Pin simulation framework, discussed in Section 5.3, allows individual fault injections to be monitored for the entire run of a benchmark, so that a determination can be made as to whether or not the fault affected final program state. Lastly, the ASIM microarchitectural framework, discussed in Section 5.4, is used to quantify the per-

formance loss associated with selected fault screeners due to pipeline flushes associated with screener false positives.

### 5.1. Fault Injection Methodology

Our fault injection points are all chosen in a random distribution over the length of the benchmark studied. The injection itself is done by altering a randomly selected bit in the nearest result register write to each selected fault injection point. In the case of the ASIM framework [5], the injection will be done at the level of individual x86 micro-ops. In the case of the Pin simulations [13], injections are performed on x86 macro-op boundaries. (Instructions in the x86 ISA are broken into micro-ops to allow for easier processing in current microarchitectures.)

### 5.2. ASIM Functional Simulation

The ASIM functional simulator is used to monitor the progression of an injected fault in the short term. The functional simulator consists of an architectural-level x86 simulator [5]. The input to the simulator is a 30 million instruction trace from each program. Ten thousand random points in each trace are identified for injection. A bit is then flipped in the result value of the nearest micro-op to each randomly selected point. After each fault is injected, the progression of that fault is monitored in detail for 5000 instructions. At the end of the 5000 instructions, a determination can be made as to whether the state of the program matches that of a corresponding clean program run. If a control flow divergence is observed from the clean state, the functional simulator searches the faulty thread and the clean thread to find a section of 1000 dynamic instructions that match. If such a section is found, the control flow divergence is said to have merged, and the state between the two runs is compared across matching instruction addresses. Since our fault screeners are designed to catch all faults, not merely harmful ones, we present overall fault detection rates. We use the ASIM functional framework to get an overview of fault effects over a large set of benchmarks.

### 5.3. Pin Simulation

Pin is a dynamic instrumentation tool that allows the user to instrument a running application [13]. The tool provides both instrumentation and analysis routines. The instrumentation routines provide hooks into the application, while analysis routines analyze aspects of the program behavior. In the Pin fault injection tool, the instrumentation routine determines where the fault is to occur and provides access to the architectural state of the application. The analysis routine uses the architectural state to insert the fault and runs the application with the faulty architectural state. Subsequent analysis routines determine how the fault impacts the eventual outcome of the program.

The advantage of using Pin is that it provides a methodology for analyzing full program behavior. Applications instrumented by Pin can run several orders of magnitude faster than ASIM’s functional simulation and, hence, billion-instruction applications instrumented by Pin can run to completion. The ASIM fault injection methodology considers a fault to be dangerous if there is a difference in memory or register state at the end of a 5K instruction window. Pin, however, only considers a fault to be dangerous if either the application crashes or there is a difference in the application output, since it runs each application to completion. We use the Pin tool to provide analysis of fault behavior over a small set of selected benchmarks run to completion.

### 5.4. ASIM Microarchitectural Simulation

The ASIM microarchitectural simulator is a detailed cycle-by-cycle model of a Pentium III-like design. This simulator is used to measure the performance impact of flushing the pipeline every time a fault screener triggers. Since only one fault is encountered during a program run, the vast majority of screener triggering will be due to detected natural perturbations.

## 6. Results

In Section 6.1, we will present the percentage of injected faults that end up being observable by a user. In Section 6.2, we will show coverage and false positive rates for all of the screeners in a fixed-size environment. In Section 6.3, we will use the full program evaluation capabilities of the Pin framework to show that a fault screener requires a resetting mechanism to keep it from losing effectiveness as the program runs for billions of instructions. We will use this information to design an implementation of a resetting invariance-based fault screener which we will present in Section 7. Unless otherwise specified, the ASIM functional framework was used to generate the data graphed. Also, unless otherwise specified, the fault screeners monitor every result register write for program perturbations (except TLB screener).

### 6.1. Long-term Fault Effects

Figure 3 shows the final result of each of 10K fault injections across six of the SPEC benchmarks. The Pin framework is used to inject faults, and the faults are monitored for the entire run of the reference input set for the benchmark. In all cases, over 50% of the faults injected do not alter the end result produced by the program. Most of the faults that do have an effect cause the program to terminate early with a non-zero exit code and accompanying signal. These represent injections that resulted in an internal error, such as a segmentation fault, within the program, and are most often caused by an address reference into unmapped space.

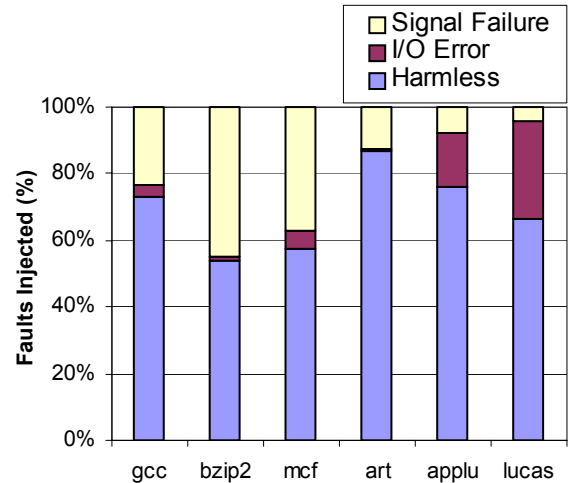


Figure 3. Results of 10K fault injections in full program runs (reference data set)

### 6.2. Fixed-size Screeners

Figure 4 shows the behavior of all screeners in a fixed-size environment on the integer and database benchmarks. Each of the screeners from Section 4 is limited to a maximum 2K-entry fault screening table, indexed in a direct-mapped manner. (Exceptions are the TLB screener which uses no additional space and the bloom screener which is 1Kbytes in total size.) The y-axis of the graph indicates how many of the 10K injected faults were identified by each screener within 64 instructions of the fault injection.

Comparing with the raw perturbation rates from Figure 1, the first thing to note from Figure 4 is the low number of faults screened on TPCC and SQL. The number of perturbations identifiable with a fixed-size table on benchmarks with large instruction footprints is dramatically less than the number that were shown to actually exist. For TPCC and SQL, the extended history and invariance screener are barely able to identify any faults at all with a fixed-size table. Some of the screeners are more vulnerable to destructive aliasing than others. In particular, the invariance screener suffers from considerable aliasing. This is because only two static instructions need map into the same location to make the invariance bit masks saturate to all bits variant. If two values map to the same invariance entry, and the values vary by a number of bits, the invariance mask will be set to indicate that all the differing bits are variant, when in fact only two values are present. The extended history screener does not perform well either, because the database workloads generate enough data values to overrun its 64 value history. The dynamic range screener does slightly better since it can represent large sets of data at once. Also interesting on the database benchmarks is the performance of the bloom-filter based screener, which is

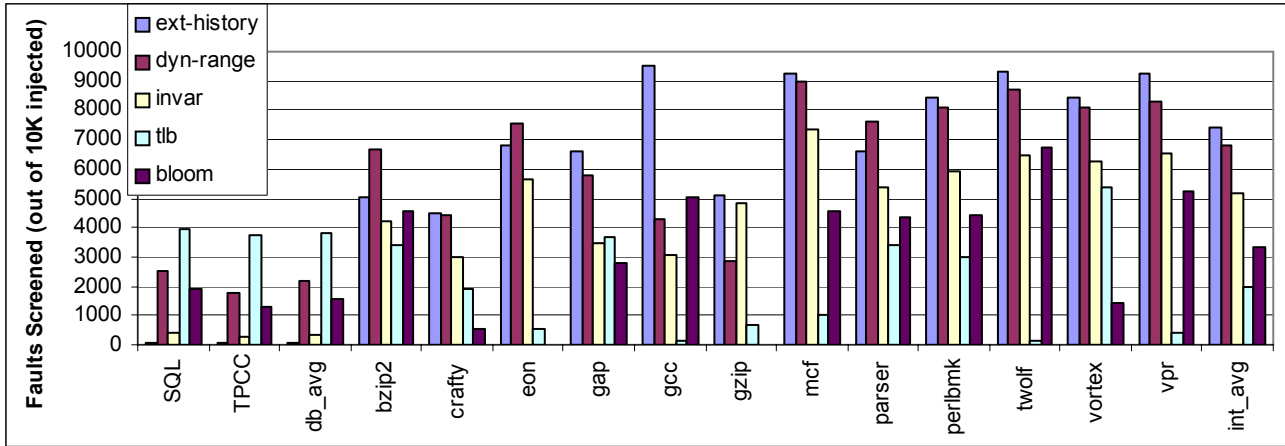


Figure 4. Fault coverage of fixed-size fault screeners (2K entry limit)

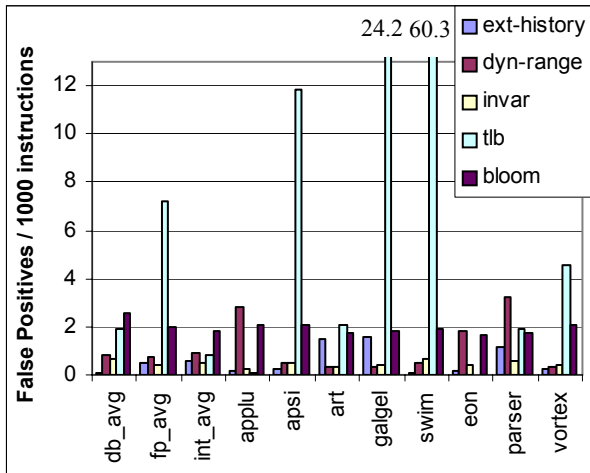


Figure 5. False positive rate for fixed-size screeners (avgs + exceptional cases)

able to detect 15% of the faults with its 1Kbyte lookup table.

Figure 5 shows the false positive rate on average for each of the benchmark suites. As previously stated, the fault screeners with filtering capability attempt to maintain an average false positive rate of one false positive per 1000 instructions or less. In addition to the averages, the graph shows any benchmarks whose false positive rate is higher than the overall average for that screener. As expected, the TLB-based screener has the highest false positive rate, up to 60.3 false positives / 1000 instructions in swim. The dynamic range screener has two benchmarks with a false positive ratio greater than 2 / 1000 instructions, applu and parser. The bloom screener averages a false positive rate of 2 / 1000 instructions in all benchmarks. We will show later in the paper that a false positive rate of 1 / 1000 instructions does not degrade performance significantly.

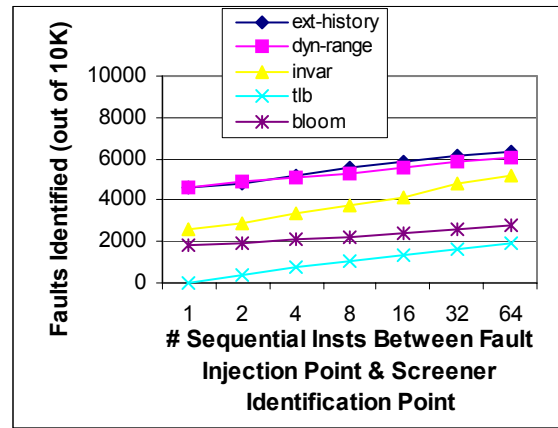
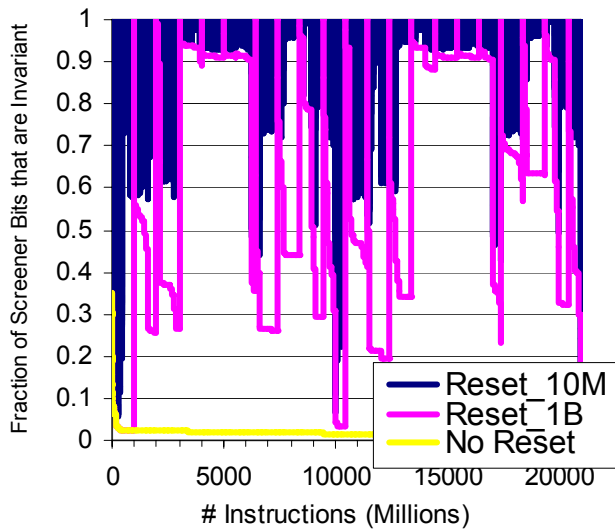


Figure 6. Average detection distance in sequential uops (integer suite)

A third metric of importance for fault screeners is detection latency. Detection latency can be critical because a perturbation may not become observable until several instructions after the fault. To be effective, a fault screener must identify the perturbation before the faulty instruction hits committed state. Figure 6 shows the average detection distance across the SpecInt suite for each fault screener type. The y-axis of the graph is the absolute number of faults detected by each screener from Figure 4. The graph shows that most of the screeners identify about 2/3 of the faults immediately, and then slowly pick up the final third with increasing distance. The extended history and dynamic range start out with the highest percentage of faults identified immediately, and the TLB-based screener, unsurprisingly, is the slowest to identify the perturbations associated with faults. The TLB can only identify faults that propagate to the high order bits of an address, so it is understandable that the screeners monitoring every instruction destination write are able to identify faults much faster. The invariance



**Figure 7. Fraction of invariant bits vs time in resetting invariance screener over a reference run of GCC**

screener also starts with a relatively lower number of faults detected immediately, and increases with a greater slope than the others. This may be because the invariance screener, like the TLB, identifies many of its faults because they propagate to the high bits of an address.

### 6.3. Effect of Resetting on Screener Behavior

Several of the fault screeners have no intrinsic aging mechanism to allow them to adapt to new program behavior. To show the need for such a mechanism, we will examine the behavior of the invariance screener on a full run of gcc using the Pin framework. Figure 7 illustrates the interaction of the invariance predictor with the various phases of gcc. The y-axis in the graph represents the fraction of all bits in the invariance screener’s tables that are currently listed as invariant. The x-axis shows the progression of gcc in millions of instructions executed. A non-resetting invariance screener is represented by the light gray line. The graph shows that gcc immediately hits a period of high variance at the beginning of the program. If the invariance screener is not reset, it quickly loses almost all of its ability to detect perturbations. If the screener is reset every 1 billion instructions, its behavior is represented by the dark gray line. This line shows that the screener recovers after the original period of high invariance but starts to collect variance bits again, culminating in a second high point of variance at a ratio of 2.6 invariant bits / 10 screener bits. The black line shows the behavior of a screener that flushes every 10 million instructions. This last screener is able to maintain more than twice as many variant bits as the screener that flushes every 1 billion instructions during the early phases of the program. There is a distinct correlation between the number of variant bits currently set in the

invariance screener and the number of perturbations that it is able to detect. Therefore, we incorporate resetting technology into our invariance screener. We will flush its tables every 10 million instructions.

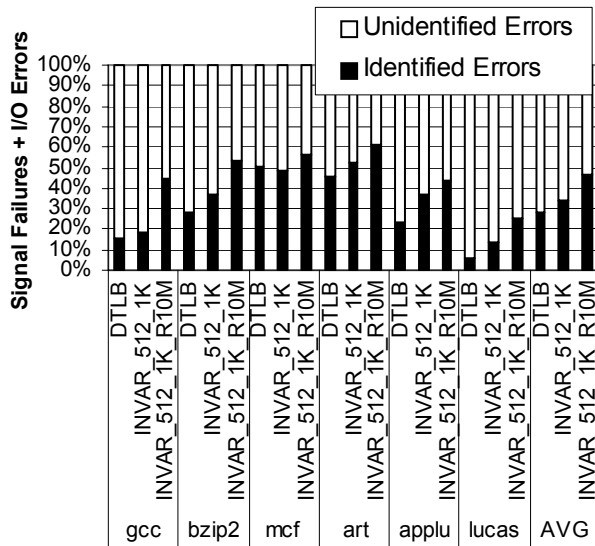
## 7. Sample Hardware Implementation

In this section, we present an example hardware implementation of one of our fault screeners. This hardware implementation is designed to plug into a current processor with as few microarchitectural modifications as possible. We choose the invariance fault screener as it has a fairly simple design and works well in the presence of regular resetting. The hardware adaptation makes several compromises to optimize the implementation to a reasonable size.

To cut down on destructive aliasing, our implementation records data only for memory instructions (i.e. loads and stores). For each static store instruction we maintain two value/delta bitmasks: one for the store address and one for the store data. Similarly, for each load instruction we maintain one value/delta bitmask for the load address (we don’t screen the values loaded from memory, since memory is assumed to be protected). For maintaining these value/delta bitmasks we use two independent storage structures, one for store/load addresses and one for store data. The value/delta bitmasks for store/load addresses are 32 bits long and for the store data are 64 bits long. In our simulation studies we use a 1K entries table for store/load addresses, and a 512 entries table for store data. Both tables are indexed with the instruction’s address and are updated speculatively on instruction execution.

Since our fault screening techniques are able to identify faults quite rapidly, we are able to use the branch recovery hardware to implement fault recovery as well. While this saves us from having to implement a checkpointing mechanism, it also reduces error coverage rates by about 30%. In order for a fault to be recoverable, the fault must be screened before any instruction commits corrupted data to the architecture state or memory. The opportunity for fault screening exists only in the relatively short amount of time an instruction spends between the execute and commit stages of the pipeline. The screening latency restrictions are architecture dependent, and based on our studies, for the x86 Pentium III based-microarchitecture we used, is limited on average to 8 instructions (average over all benchmark sets). Hence, for the purposes of this analysis, we will only consider a fault to be successfully identified by a fault screener if the fault’s perturbation is observed before 8 consecutive instructions have gone by.

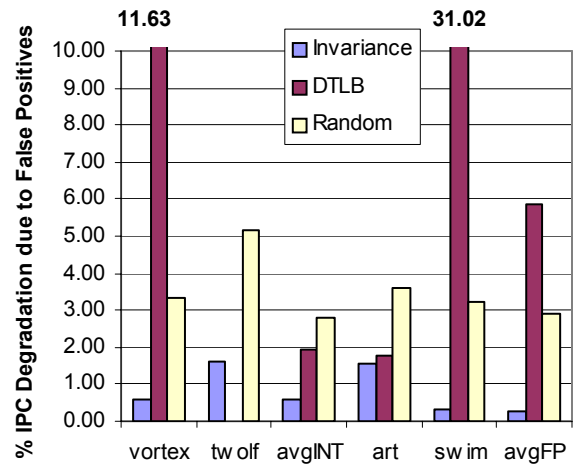
Figure 8 shows the coverage of the invariance detector compared to the performance of a TLB-based fault screening mechanism. The y-axis of the figure starts with the number of faults that resulted in long-term program errors, either signal failures or incorrect program output. The black



**Figure 8. Error coverage of realistic invariance screener implementation (detection within 8 instructions of fault)**

bar represents the percentage of these error-causing faults that are caught by each of the fault screener types. The first fault screener listed is a TLB screener, which triggers and flushes the pipeline every time there is a data TLB miss. The second screener, “INVAR\_512\_1K,” represents a non-resetting invariance-based screener with 1K entries for store/load addresses and 512 entries for store data. The third screener, “INVAR\_512\_1K\_R10M,” represents the same invariance-based screener where the screener is completely reset every 10 million instructions. The graph shows that on average, the realistic invariance-based detector can detect almost 50% of silent data corruption errors within 8 instructions of the point that the fault first propagated to speculative state. If we allowed faults to be identified within 100 instructions of the fault injection point (not shown) the average coverage rate of silent data corruption errors rises to over 70%. This data was generated over by the Pin framework and represents coverage rates over the full run of the reference set of each benchmark.

Figure 9 shows the performance degradation due to the requirement that a pipeline flush occur every time a fault screener triggers. The suite averages and worst performing benchmark for each screener are shown. This data was generated with the ASIM full microarchitectural simulator modeling a Pentium III-like design. Representative traces of the benchmarks were run for a minimum of 10 million instructions. The y-axis on the graph represents the performance degradation in percent for a Pentium III-like processor incorporating an active fault screener compared to the same processor without a fault screener. The graph shows that on average, a TLB-based fault screener causes a perfor-



**Figure 9. IPC degradation due to screener false positive flushing**

mance degradation of 2% in the integer benchmarks and almost 6% in the floating point benchmarks, while the realistic invariance-based fault screener degrades performance on average less than 1% in both benchmark suites. Interestingly, a mechanism that randomly flushes the pipeline at a rate comparable to the invariance-based fault screener (1 flush / 1000 instructions) harms performance much more than the invariance-based fault screener (an average of 2.8% on the integer and 2.9% on the floating point benchmarks) The invariance-based fault screener has better performance than the random mechanism because a large fraction of its flushes occur on the wrong path (55% on average vs 20% in the random case). This is because the invariance screener is more likely to see anomalous data on the wrong path. False positive flushes on the wrong path are less harmful.

## 8. Summary

This paper makes several contributions. First, we show that the presence of a program value perturbation is a reliable indicator of the presence of a fault. We show that over 85% of injected faults on the SpecInt suite and over 75% of faults overall cause perturbations detectable by an idealized algorithm. This level of fault detection could potentially allow a design to meet its soft error goals without resorting to extreme measures. We show that, on average over the SpecInt & SpecFP benchmarks, the incidence of detectable perturbations increases by more than a factor of 30 in the presence of a fault compared to the level of perturbations in the same set of dynamic code segments in the absence of a fault.

Secondly, we analyze the performance of screeners based on value history, dynamic range, bit-invariance, and a bloom-filter and show the strengths and weaknesses of each. We compare these results against other implementa-

tions of fault screeners in related work, both from the hardware and software communities.

Thirdly, we monitor the propagation of architectural state faults in detail both in the short term via a microarchitectural simulator, and over a full program run via a binary modification tool. This allows us to examine both the immediate effects of faulty data on dependent surrounding instructions and the long-term user-visible effects that the faulty data has, such as causing a segfault or changing the output of the program.

Lastly, we present a sample hardware implementation of an invariance-based screener that works off of existing branch misprediction recovery mechanisms. We show that nearly half of the faults that would have eventually caused an output error or early program termination can be caught and repaired while still in processor speculative state. The performance loss due to flushing the pipeline on all false positives is less than 1% for our implementation of this screener.

## References

- [1] D. Armstrong, H. Kim, O. Mutlu, and Y. Patt, "Wrong Path Events: Exploiting Unusual and Illegal Program Behavior for Early Misprediction Detection and Recovery," *37th International Symposium on Microarchitecture (MICRO)*, December 2004.
- [2] Anderson, J. "Computer security threat monitoring and surveillance", *Technical Report James P. Anderson Company*, Fort Washington, Pa, 1980
- [3] B. Bloom. "Space / time trade-offs in hash coding with allowable errors," *Communications of the ACM*, Volume 13, Issue 7, pp. 422-426, July 1970.
- [4] D. Denning, "An intrusion-detection model," *IEEE Transactions on Software Engineering*, vol 13, number 2, pp. 222-232, February 1987.
- [5] J. Emer, P. Ahuja, N. Binkert, E. Borch, R. Espasa, T. Juan, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, and S. Wallace, "Asim: A Performance Model Framework," *IEEE Computer*, 35(2):68-76, Feb. 2002.
- [6] D. Engler, D. Chen, A. Chou, S. Hallem, B. Chelf. "Bugs as Deviant Behavior: A General Approach to Inferring Errors in System Code," *Proceedings of Symposium on Operating System Principles*, October 2001.
- [7] A. Ghosh, A. Schwartzbard, and M. Schatz. "Using program behavior profiles for intrusion detection", *Proceedings of the SANS Intrusion Detection Workshop*, 1999.
- [8] S. Hangal, M. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," *Proceedings of the International Conference on Software Engineering*, ICSE'02, May 2002.
- [9] M. Lipasti, C. Wilkerson, and J. Shen, "Value Locality and Data Speculation," *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147, October 1996.
- [10] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walstra, and C. Dai, "Impact of CMOS Scaling and SOI on soft error rates of logic processes," *VLSI Technology Digest of Technical Papers*, 2001.
- [11] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Borkar, "Scaling trends of Cosmic Rays induced Soft Errors in static latches beyond 0.18 $\mu$ ," *Symposium on VLSI Circuits Digest of Technical Papers*, 2001.
- [12] S. Patel, Z. Kalbarczyk, R. Iyer, W. Magda, N. Nakka, "A Processor-Level Framework for High-Performance and High-Dependability," *Proceedings of the Workshop on Evaluating and Architecting Systems for Dependability*, 2001.
- [13] C.K. Luk, R. Cohn, R. Muth, et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 190-200, Chicago, IL, 2005.
- [14] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, May 2002.
- [15] G. Reis, J. Chang, N. Vachharajani, S.S. Mukherjee. "Design and Evaluation of Hybrid Fault Detection Systems," *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, June 2005.
- [16] M. Roesch. "Snort - lightweight intrusion detection for networks," *13th Systems Administration Conference*, pages 229-238, November 1999.
- [17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems* Vol15, No. 4. pp. 391-411, November 1997.
- [18] N. Wang, M. Fertig, S. Patel. "Y-Branches: When You Come to a Fork in the Road, Take It," *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, New Orleans, LA, September 2003.
- [19] N. Wang, S. Patel. "ReStore: Symptom Based Soft Error Detection in Microprocessors," *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, Yokohama, Japan, June 2005.
- [20] C. Weaver, J. Emer, S.S. Mukherjee, and S. Reinhardt. "Techniques to Reduce the Soft Error Rate of a Microprocessor," *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004