

# Speculative Instruction Validation for Performance-Reliability Trade-off

Sumeet Kumar  
Electrical and Computer Engineering  
Binghamton University  
Binghamton, NY 13902  
skumar1@binghamton.edu

Aneesh Aggarwal  
Electrical and Computer Engineering  
Binghamton University  
Binghamton, NY 13902  
aneesh@binghamton.edu

## Abstract

*With reducing feature size, increasing chip capacity, and increasing clock speed, microprocessors are becoming increasingly susceptible to transient (soft) errors. Redundant multi-threading (RMT) is an attractive approach for concurrent error detection. RMT provides complete error coverage, while incurring a significant performance impact because of the redundant thread. Achieving perfect reliability at the expense of a high performance drop is not a good design option for systems where slight vulnerability may still achieve the desired error rates.*

*In this paper, we explore speculative mechanisms to trade-off reliability for performance in RMT. Our basic approach validates the execution of an instruction by comparing its result against the expected result. Only those instructions are redundantly executed for which the validations fail. This mechanism is expected to have a minimal vulnerability impact because it is highly unlikely that an erroneous result matches the expected value. We also propose several extensions to the basic approach that further explore the performance-reliability trade-off design space. A combination of these techniques incur about 10% performance impact and about 0.09% undetected base error rate, compared to about 25% performance impact for RMT with no undetected errors.*

**Keywords:** Concurrent Error Detection, Reducing Instruction Redundancy, Redundant Multi-threading, Instruction Validation, Performance-Reliability Trade-off

## 1 Introduction

With the current trends in transistor size, voltage, and clock frequency, microprocessors are becoming increasingly susceptible to hardware failures. A large portion of the errors in the current technology are soft errors [4,22] that randomly flip the bit values in the processor. These errors are temporary in nature and are particularly troublesome because they elude most of the current testing methods. A popular approach to detect soft errors is *Redundant Multi-threading (RMT)*, where a thread is executed redundantly and errors are detected by corroborating the results from redundant executions. There are various different implementations of RMT [1, 4, 6, 9, 16, 18, 20, 21, 23, 27, 28, 31].

One popular RMT implementation is *SRT*, where the redundant threads execute entirely independently of each other in a simultaneous multi-threaded (SMT) environment [6, 20, 21, 28]. Redundantly executing every instruction in an application for complete error coverage results in signifi-

cant performance degradation. For instance, our experiments show that SRT, our SRT architecture is explained in detail in Section 2, results in about 25% reduction in the average IPC of the simulated SPEC2K benchmarks. For many systems, achieving perfect reliability at the expense of such high performance impact is not a good design option, especially if a slightly vulnerable system may still achieve the desired error rates. Hence, it is essential to explore techniques that achieve a good performance-reliability trade-off [7].

Recently, there have been a few proposals, such as Re-Store [30], PB [17], and LB [5], that do not perform any redundant execution. These approaches flag errors if perturbations are observed in the program behaviors such as branch predictions, instruction results, TLB misses, etc. The false positive rate for these schemes can be high if the program behavior changes correctly. Similarly, the false negative rate for these schemes can also be high. For instance, these schemes will miss errors in instructions that do not have predictable behavior and/or do not lead to the behavior that is heeded. *It is important to note that these approaches may still have a high performance impact because of reverting back on every false positive.*

One method to mitigate the performance impact of redundancy is to avoid redundant execution of as many instructions as possible. There are two different approaches that have been proposed to reduce redundancy in RMT.

The first set of proposals, such as PER-IRTR [7], VCSR [26], and RMT-Toggling [29], react to the processor state in order to reduce redundancy in RMT. For instance, PER-IRTR avoids redundant execution during high IPC program phases. Similarly, VCSR performs redundant executions only when the AVF of certain processor structures increases beyond the acceptable limits. RMT-Toggling uses AVF prediction to identify low AVF program phases, and avoids redundant execution for those phases. In these approaches, any error that may affect the program outcome may go undetected during the unprotected phases. This may result in a high false negative rates, especially in those approaches, such as PER-IRTR, that give priority to performance. Similarly, approaches that give priority to reliability, such as VCSR [26] and RMT-Toggling [29] may have a high performance impact.

A second set of approaches, such as SlicK [14], SS-mod [19] (modified SlipStream [27]), DIE-IRB [15], proac-

tively avoid redundant execution of those instructions that can be validated through means other than redundant execution. Slick compares the addresses and the data values of store instructions with those of their previous instances. If a comparison matches, then the entire slice of instructions that lead to the production of the store instruction's address and data is not executed redundantly. Similarly, SS-mod removes backward slices of silent stores [11], dead values [3, 13], and silent writes. Note that SS-mod uses a slipstream execution model. However, their techniques can also be applied to a typical RMT execution model. The design complexity of these schemes is significantly higher because highly complex hardware resources are required to form backward slices. DIE-IRB uses the instruction reuse [25] concept to avoid redundant execution of instructions that have the same operand and result values as adapted for previous instances. The false positive rate of SS-mod (in SRT) and DIE-IRB is zero, but Slick may have some false alarms. False alarms may be raised in Slick because the values produced in a non-executing redundant slice cannot be forwarded to an executing redundant slice.

All the three approaches in the second set may also miss a considerable number of redundancy reduction opportunities. Slick misses avoiding redundant execution of highly predictable instructions that form a part of unpredictable slices. It will rarely happen that a part of the slice is unpredictable but the entire slice is predictable, in which case Slick may benefit. SS-mod misses redundancy reduction for any instruction with predictable behavior that does not fall in the monitored categories. For instance, when capturing silent writes, it misses redundancy reduction if different static instructions write different values, where the values are repeating for each individual static instruction, to the same architectural register. DIE-IRB also misses reducing redundancy for instructions that produce predictable, but variable result values such as strided values. Furthermore, DIE-IRB only focuses on saving the ALU bandwidth. Hence, it may still have a higher performance impact due to bottlenecks in other resources.

The approach in [10] reduces instruction redundancy only in decoupled RMTs [24], and requires that many resources, such as front-end pipe stages, register file, etc., are protected using error codes.

In this paper, we explore a speculative approach – Speculative Instruction Validation (*SpecIV*) – where an instruction's execution is validated by comparing its result against the expected value. Only those instructions whose validations fail are redundantly executed. Spec-IV belongs to the second set of approaches. SpecIV avoids redundant execution of any instruction that demonstrates predictable behavior, achieving higher levels of redundancy reduction. It avoids false positives altogether and has a very low false negative rate because of the high unlikelihood of an erroneous result matching the expected value.

The specific contributions of this paper are:

- A speculative instruction validation technique that achieves considerable reduction in the performance impact of SRT, while incurring only a slight increase in

processor vulnerability. Our experiments show that the base SpecIV technique reduces the performance impact of SRT by an average of about 54%, while increasing the undetected error rate from zero to 0.45%.

- Several extensions to the base SpecIV technique that explore the performance-reliability trade-off design space. A combination of performance-improving techniques with low reliability impact and reliability-improving techniques with low performance impact further reduce the performance impact of SRT by an average of about 60%, while reducing the undetected error rate to only about 0.09%.
- Several innovative extensions of SpecIV that further trade-off larger amount of reliability for significantly better performance, as high as an average of about 80% reduction in the performance impact of SRT.

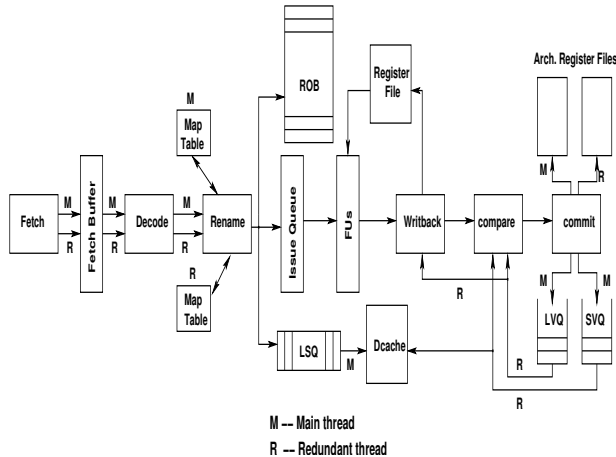
The rest of the paper is organized as follows. Section 2 discusses the base SRT model. Section 3 presents the details of SpecIV. Section 4 presents the experimental results and analysis for SpecIV and compares it against SS-mod and DIE-IRB. Section 5 explores the performance-reliability design space. We conclude in section 6.

## 2 Background

Figure 1 shows the schematic diagram of our SRT model. All connections are not shown in the figure for better legibility. In this model, a thread is executed redundantly for error detection. The "redundant" thread runs behind the "main" thread by a few instructions (*staggered* execution) [28]. Studies have shown that a *staggered* SRT model performs better than running both the threads in a *lock-stepped* manner [6, 24]. This is because the trailing thread does not incur many of the branch misprediction and the load miss penalties incurred by the leading thread. In this model, each thread has a private PC register, rename map table, and architectural register file, whereas the rest of the resources are shared.

The two threads are executed concurrently. At commit, the two threads update their respective architectural register files. The main thread instructions commit much before their redundant thread counterparts because of the slack. When the main thread store instructions commit, they also write their addresses and store values at the tail of a circular FIFO Store Value Queue (SVQ). The store instructions do not update the memory at this time. When the redundant thread store instructions commit, they compare their addresses and store values with those of their main thread counterparts in the SVQ. On any failed comparison in the SVQ, an error is flagged.

Only the main thread load instructions access the memory, which is assumed to be transient fault tolerant (by using Error Detection and Correction Codes). The values thus loaded are also forwarded to the redundant thread using a circular FIFO Load Value Queue (LVQ). However, the redundant thread still needs to verify the load addresses. For this, the main thread load instructions also write their addresses in the LVQ. When a redundant thread load instruction commits, it also compares its address with the head of the LVQ. The



**Figure 1. Schematic Diagram of a SRT Redundant Multi-threading Processor**

branch outcomes for the main thread are also forwarded to the redundant thread.

When an error is detected, the execution is rolled back to the checkpoint. For this, a checkpoint is created at regular intervals. To create a checkpoint, the architectural register file values of the two threads are compared, and if they match, they are check-pointed. Integrity of checkpoints is maintained by not allowing the dirty data to propagate to the memory between checkpoints [12].

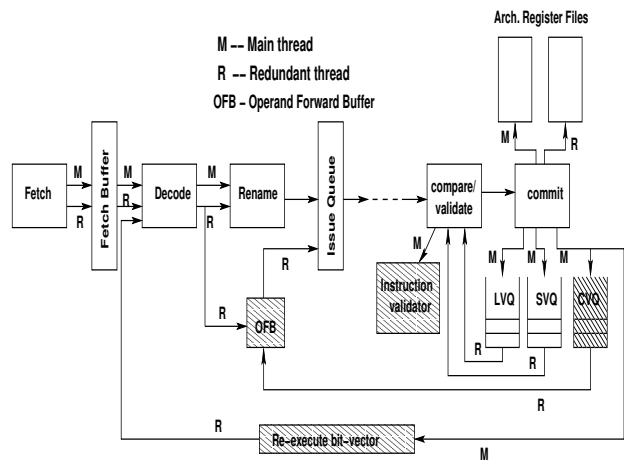
### 3 Base Performance-Reliability Trade-off Technique

The basic idea of our approach is to validate instructions' execution by comparing their results to the expected values. We refer to this technique as Speculative Instruction Validation (*SpecIV*). Our technique validates the addresses generated by load and store instructions, the branch outcomes for the branch instructions, and the register results for the remaining instructions. To validate the results, we maintain an *instruction validator* that stores the expected results of instructions. The expected results of memory instructions are the memory addresses generated by them. If an instruction's result is the same as the expected result, then it is not redundantly executed. Otherwise, the instruction is redundantly executed. The branch predictor serves as the validator for control instructions. A store instruction is redundantly executed if the validation of the store address and/or the store data fails. The vulnerability impact of SpecIV is discussed in Section 3.2.

#### 3.1 Implementation Details

Figure 2 shows the schematic diagram of the processor with the SpecIV technique. The additional structures required in SpecIV are shaded in Figure 2. The processor core remains the same and hence is removed for better legibility. The processor is provided with a circular FIFO *re-execute* bit-vector. The committing main thread instructions update the bits in-order at the tail of the bit-vector (tail-bits). The

fetched redundant thread instructions consume the bits in-order at the head of the bit-vector (head-bits). If the validation of a main thread instruction succeeds, then the tail-bit is reset, else the tail-bit is set. All redundant thread instructions are fetched, but they are executed only if their re-execute bits are set. This mechanism differs slightly for the store instructions, as discussed later in the section. In SpecIV, the re-execute bit-vector is a highly critical structure and may be replicated for better protection. The number of bits in this bit-vector is larger than the slack between the two threads. The commit of main thread stalls if the bit-vector is full, and the decode of redundant thread stalls if the bit-vector is empty. This ensures that a main thread instruction commits before its redundant counterpart is sent into the pipeline.



**Figure 2. Schematic Diagram of the SpecIV Technique; Additional Microarchitectural Structures are shaded**

We experiment with a stride-based instruction validator<sup>1</sup>. Branch instructions are not provided entries in the validator. Each entry of the validator consists of a *PC-tag*, the *last value*, the *stride value* and a *stride-valid bit*. The stride-valid bit is "1" if a stride is observed, else it is "0". The expected result from the validator is the *lastvalue* + *stride* if the stride-valid bit is set or the *lastvalue* if it is reset. In parallel to transferring the main thread instructions' results to the architectural register file at commit, they are validated against the expected values. For memory instructions, the addresses from the load/store queue are validated against the expected values. If an instruction's validation fails, then the last value is updated with its result and the stride-valid bit is reset. If an instruction does not have a validator entry, it is allocated a least recently used (LRU) entry for a set-associative validator.

In SpecIV, some redundant thread instructions update the architectural RF whereas others do not. This results in false positives when the architectural states of the two threads are compared at checkpoint creation. To avoid false positives

<sup>1</sup>We also experimented with last-value validator. Its performance was lower than that of a stride-based validator because the stride-based validator also captures the validations from the last-value validator. Hence, we only present the results of a stride-based validator.

and to prevent errors in the architectural registers that have not been redundantly generated, these registers are marked and protected through parity bits in the main thread’s architectural register file. The marked architectural registers are not compared at checkpoint creation.

An *operand forward buffer (OFB)* and a *FIFO Commit Value Queue (CVQ)* are used to forward the results of non-executing redundant instructions to the executing ones. The OFB has one entry for each architectural register. The register results of main thread instructions with successful validations at commit are written at the tail of CVQ. In decode, a non-executing redundant instruction marks the OFB entry for its destination register as valid, and transfers the value from the CVQ-head into that OFB-entry. An executing redundant instruction invalidates its destination register’s OFB-entry. Note that a successfully validated load instruction writes the loaded value in the CVQ and does not write anything in the LVQ.

The executing redundant instructions that are dependent on valid OFB entries read their source operand values from the OFB. The accesses to CVQ and OFB are not on the critical path because the values are not required till these instructions are dispatched. This approach increases the width of the scheduler payload RAM (to include the register operand values) for processors where operands are read from the register file after the instructions are issued for execution. Alternately, the executing redundant instructions can also read their non-produced operand values directly from the CVQ. However, this option brings the CVQ access on the critical path and limits its size.

A store instruction is not re-executed only if its re-execute bit is reset (*i.e.* store address has been successfully validated) and the OFB entry for its store register is valid (*i.e.* store value has been successfully validated).

### 3.2 Vulnerability Impact of SpecIV

Soft errors result in Silent Data Corruption (SDC) if they are not detected and they update the processor state incorrectly. Of course, an error may not result in faulty execution if it does not propagate to an instruction [31] or is masked because of branch mispredictions, silent stores [11], dead values [13], etc. In SpecIV, an unmasked instruction error may go undetected only if an incorrect result (due to a soft error) produced by a main thread instruction matches the expected result, thus preventing the execution of the redundant counterpart of the instruction. The vulnerability of SpecIV is expected to be minimal, as corroborated in Table 5, because it is highly unlikely that an erroneous result matches the expected value.

Errors in CVQ and OFB are always detected because they are only used by the redundant thread. Errors in the validator affect the program outcome, only if they are accompanied with specific errors in instructions’ execution that result in successful validations from a faulty validator. Such scenarios will not occur in single event upsets. To protect against errors in SVQ entries occupied by non-executing store instructions, parity bits are generated for non-executing store instructions’ addresses. Parity bits are already generated for stored values

in the original SRT implementation. In SpecIV, the parity bits are generated before writing the values to the SVQ.

## 4 Experimental Results

### 4.1 Experimental Setup

The hardware parameters for the base superscalar processor are given in Table 1. Our superscalar model consists of separate integer and floating point subsystems. We use a modified SimpleScalar simulator [2], simulating a 32-bit PISA architecture. We present the results for a representative set of 15 SPEC2000 benchmarks (*vpr, mcf, parser, bzip2, vortex, gcc, ammp, equake, applu, art, apsi, mgrid, mesa, swim, and wupwise*) to save space. The statistics are collected for 500M instructions after skipping the first 1B instructions. We use a slack of 128 instructions among the two threads, and a 4K-entry direct mapped stride-based instruction validator, with a 16-bit wide stride value field stored in 2’s complement.

### 4.2 Performance Results

Table 2 shows the percentage of instructions (divided into four categories: store, result-producing integer, branch, and floating-point) saved from redundant execution. The percentage for each category is out of the total instructions in that category. The weighted average weighs each percentage by the number of instructions in that category for each individual benchmark. The store instruction measurements in Table 2 are for only those that are entirely dropped from the pipeline. The savings for branch instructions depend on the branch prediction accuracy. The table also presents the percentage of the total instructions that avoid redundant execution. *Nan* values imply no instructions in that category.

The percentage of instructions saved from redundant execution depends on the percentage of instructions whose results match the expected values. On an average, about 61% of instructions are removed from redundant execution. A higher redundancy reduction is observed for integer instructions than floating-point instructions. This is expected because integer instructions’ results, which also include load addresses, are generally more predictable using a stride-based approach.

Table 2 compares SpecIV with SS-mod and DIE-IRB. SS-mod was originally proposed for a Slipstream processor. However, we adapt SS-mod for our SRT model. Furthermore, backward slices are not formed for any technique. Table 2 shows that SpecIV saves significantly more instructions from redundant execution than SS-mod and DIE-IRB.

Reducing instruction redundancy improves performance by reducing the pressure on various resources such as the ROB, the load/store queue, the register file, the issue queue, the issue width, etc. Figure 3 shows the IPC results of SpecIV compared to base SRT, single-threaded execution, SS-mod, and DIE-IRB. We observed a 25% reduction in the average IPC for the base SRT model. SpecIV recovers, on an average, about 51% of the performance loss. Some benchmarks, *e.g.* *mcf* and *parser*, recover a large portion of the performance loss, whereas some benchmarks, *e.g.* *wupwise* and *applu* recover a small portion. This is because the performance loss recovered depends on the bottlenecks created by

Parameter	Value	Parameter	Value
Fetch/Decode/Commit Width	8 instructions	FP FUs	3 ALU, 1 Mul/Div
Phy. Register File	128 INT/128 FP entries, 1-cycle inter-subsystem lat.	Int. FUs	3 ALU, 2 AGU, 1 Mul/Div
Issue Width	5/3 INT/FP instructions	Issue Queue	48 INT/32 FP Instructions
Branch Predictor	Bimodal 4K entries	BTB Size	4K entries, 2-way assoc.
L1 - I-cache	32K, direct-map, 2 cycle latency	L1 - D-cache	32K, 4-way assoc., 2 cycle latency, 2 r/w ports
Memory Latency	100 cycles first word 2 cycle/inter-word	L2 - cache	unified 512K, 8-way assoc., 10 cycles
ROB size	164 entries	LSB size	40 load/ 40 store entries

Table 1. Baseline Processor Hardware Parameters for the Experimental Evaluation

	Store	Integer	Float	Branch	Total	SS-mod	DIE-IRB
vpr	12	35	20	83	43	35	34
gcc	94	81	9	87	90	24	27
mcf	48	48	nan	88	63	45	38
parser	45	51	nan	81	63	43	52
vortex	29	27	nan	85	42	25	33
bzip2	22	55	nan	88	61	38	42
wupwise	23	24	33	100	38	37	32
swim	28	46	36	98	50	34	45
mgrid	52	97	49	99	72	40	13
applu	10	62	14	89	33	37	14
mesa	49	62	nan	90	71	36	55
art	57	92	76	94	87	50	42
equake	44	48	35	85	55	35	37
ammp	77	82	20	98	84	40	48
apsi	40	55	77	86	63	41	32
Weighted Average	47	56	40	86	61	37	35

Table 2. Percentage instruction redundancy reduction in different categories of instructions

the redundant instructions in the different benchmarks. The IPC of SpecIV is about 8% better than SS-mod and about 11% better than DIE-IRB.

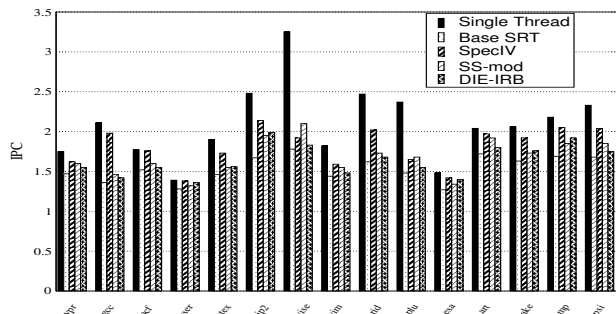


Figure 3. IPC Results

### 4.3 Vulnerability Impact Measurements

We assume a single event upset fault model in which only a single bit can get corrupted any time. We evaluate vulnerability by measuring the undetected instruction error rate, *i.e.* errors that result in faulty execution of committing instructions divided by errors that propagate to instructions. When an error propagates to an instruction, it may affect the control

path or the data path. We only evaluate the vulnerability for errors in the data path. In the data path, an error may occur in the register identifiers or the data values. Table 3 categorizes different processor structures in the data path that contribute to errors in the register identifiers and the data values. For instance, errors in the register files, operand forwarding data-path, and the immediate value fields of the instruction, result in errors in the instructions' operand values.

In our experiments, we randomly inject errors ?? in the error fields shown in Table 3 with a mean time between errors of 8 cycles. To inject an error, we flip a random bit in the error field of a random instruction. These experiments are conducted for both SpecIV, Single threaded execution, SS-mod, and DIE-IRB. Original SRT has zero vulnerability. The effects of each error are recorded separately. An instruction error is counted as undetected if that instruction commits and its result is different from the correct one and matches the expected result from the validator (for SpecIV). Our evaluation does not mask errors due to dynamically dead instructions and silent stores. If this masking is considered, then the undetected error rates will reduce by similar percentages in all the techniques.

Table 4 presents the undetected instruction error rate of

Error Field	Processor Resources Covered
Source Arch. Reg.	arch. RF decode, rename table decoder, front-end pipe stages
Source Physical Reg.	rename table entries, IQ source fields, rename-dispatch pipe stages, register file decoder
Operand Value	register file entries, arch. register file entries, operand forwarding data-path, instructions immediate value field
Result Value	functional units, load/store queue, execution-writeback data-path
Destination Arch. Reg.	front-end pipe stages, ROB fields, rename table decoder, arch. RF decoder
Destination Phys. Reg.	rename table update data-path, rename-dispatch pipe stages, IQ dest. fields, ROB fields register allocator, register file decoder

**Table 3. Processor coverage of the various errors**

the single thread execution and SpecIV. The processor vulnerability due to each field is measured separately. We observed that the errors in the destination physical registers were detected by deadlocks or allocations of a busy register and/or deallocation of a free register. Hence, we do not show the results for destination physical register errors. Table 4 shows that SpecIV has a negligible vulnerability to errors, compared to the single threaded execution. For instance, percentage of undetected result value errors reduces from about 95% for a single thread to almost zero for SpecIV. Hence, instead of perfect reliability with 25% performance impact in SRT, base SpecIV achieves only about 12% impact with 0.45% vulnerability to errors. DIE-IRB has zero vulnerability because it requires at least two upsets for an error to go undetected. SS-mod had an undetected instruction error rate of about 0.28%.

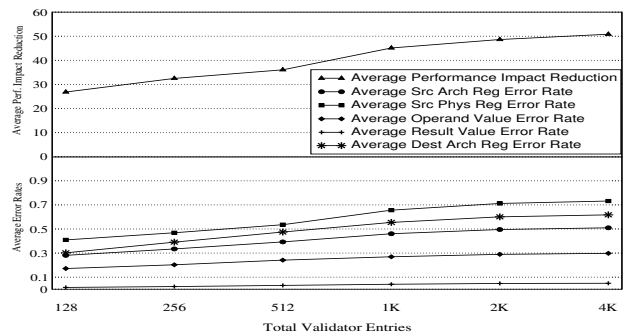
To further understand the vulnerability impact of SpecIV, Table 5 shows the percentages of errors that are either masked due to branch misprediction or masked due to correct result generation by faulty instructions or detected by the validator. Table 5 shows the percentages for only the errors in the main thread. Errors propagating to the redundant thread are immediately detected. Table 5 shows that the extremely low vulnerability of SpecIV is primarily due to the detection of errors by the validator. The errors masked due to correct results by the faulty instructions are mainly the errors in branch instructions, where a faulty branch instruction still follows the correct control flow path. We do not inject errors in the single-bit branch outcomes. Hence, the correct result masking for result value errors is *zero*.

#### 4.4 Sensitivity to Validator Size

A 4K direct-mapped validator requires about 34 KB storage space. Figure 4 plots the average IPC impact reduction and the average undetected instruction error rates for different instruction validator sizes. We only present results for direct mapped validators. The performance of set-associative

validators was as expected. For instance, 256-set 4-way and 512-set 2-way set associative validators performed very similar to the direct mapped 1K validator. IPC impact reduction increases when the validator size increases, saturating at about 2K entries, because larger validators store the history of more instructions leading to more successful validations. Similarly, undetected error rates increase with the validator size because more validations increase the number of successful validations of incorrect results.

Figure 4 shows that a 2K-entry validator, requiring only about 16 KB storage space, may be sufficient to capture all validations. Henceforth, we only present the results with a 2K-entry validator. We further experimented with a validator that stored a 10-bit PC tag (instead of a 18-bit PC tag) and a 8-bit stride value (instead of a 16-bit stride value). A 10-bit PC tag may result in some interference between instructions that map to the same entries. The storage overhead of a 2K-entry validator with 10-bit PC tag and 8-bit stride value is about 12KB; the performance of SpecIV reduces by about 0.5%, while the average undetected instruction error rate increases from 0.45% to 0.46%.



**Figure 4. Sensitivity of Error Rate and Performance Impact Reduction to the Validator Size**

#### 4.5 Conserving Fetch-to-Decode Bandwidth

In SpecIV, once the slack between the two threads is satisfied, equal number of main and redundant thread instructions are forwarded to decode, even though many of the redundant thread instructions are dropped after decode. The dropped redundant instructions only require their destination register identifiers to update the OFB. To avoid the dropped redundant instructions from using the decode bandwidth, we use a FIFO *destination id queue (DIQ)*. Each committing main thread instruction either invalidates the tail DIQ entry if it has to be redundantly executed or places its architectural destination register identifier in the tail DIQ entry. Each fetched redundant thread instruction consumes the head DIQ entry. The redundant thread instructions with valid DIQ entries simply update the appropriate OFB entries (using the destination identifiers from the DIQ) with the values from CVQ. The additional fetch-to-decode bandwidth thus made available is used by other main thread instructions. This approach does not affect the decode width and the OFB and CVQ access bandwidths. Average reduction in performance impact for SpecIV with 2K-entry validator increases to about

	Single Thread / SpecIV					
	Arch. Source	Phys. Source	Operand Value	Result Value	Arch. Dest.	Total
vpr	46.79 / 0.62	50.38 / 1.06	70.01 / 0.43	89.41 / 0.06	55.99 / 0.77	59.84 / 0.59
gcc	77.87 / 0.21	77.07 / 0.46	89.92 / 0.29	98.31 / 0.01	49.17 / 0.49	80.78 / 0.25
mcf	43.08 / 0.91	45.86 / 1.5	61.23 / 0.92	94.86 / 0.08	40.57 / 0.85	52.74 / 0.86
parser	44.32 / 0.64	47.79 / 1.32	62.14 / 0.65	84.82 / 0.05	45.74 / 1	53.84 / 0.7
vortex	69.04 / 0.19	69.04 / 0.49	79.2 / 0.31	97.34 / 0.03	53.43 / 0.31	72.96 / 0.25
bzip2	62.02 / 0.84	69.61 / 0.98	68.72 / 0.54	95.54 / 0.09	68.2 / 0.67	70.91 / 0.65
wupwise	89.3 / 0	87.51 / 0	92.35 / 0	100 / 0	98.26 / 0.11	92.76 / 0.02
swim	82.9 / 0.43	79.16 / 0.51	81.61 / 0.12	99.16 / 0.06	74.28 / 0.32	84.39 / 0.28
mgrid	96.19 / 0.03	96.67 / 0.01	97.5 / 0.02	99.98 / 0.01	94.16 / 0	96.99 / 0.02
applu	95.21 / 0.28	94.93 / 0.24	91.93 / 0.07	99.99 / 0.01	95.16 / 0	95.53 / 0.12
mesa	42.38 / 0.29	42.05 / 0.34	50.04 / 0.15	73.93 / 0.06	39.01 / 0.77	47.74 / 0.35
art	63.44 / 0.5	65.14 / 0.51	88.28 / 0.21	98.31 / 0.05	71 / 0.33	76.85 / 0.33
equake	54 / 1.89	63.47 / 2.84	78.41 / 0.71	94.93 / 0.2	59.56 / 1.66	67.51 / 1.48
ammp	68.32 / 0.2	75.54 / 0.57	68.06 / 0.16	99.71 / 0.04	83.59 / 0.71	76.93 / 0.27
apsi	70.12 / 0.49	76.05 / 0.66	80.55 / 0.31	96.98 / 0.07	68.53 / 1.14	77.31 / 0.55
Average	67 / 0.5	69.35 / 0.77	77.33 / 0.33	94.89 / 0.05	66.44 / 0.61	73.8 / 0.45

Table 4. Percentage undetected instruction errors for single-thread and SpecIV

54% by conserving the fetch to decode bandwidth, while maintaining the average total undetected error rate at 0.45%.

## 5 Performance-Reliability Trade-offs

In this section, we present various techniques that further explore the performance-reliability trade-off design space using SpecIV as the base technique. Figure 5 categorizes the various techniques explored for performance-reliability trade-off. The techniques either trade-off performance for better reliability or trade-off reliability for better performance. When trading-off performance for reliability, we only explore techniques that have a low performance impact. When trading-off reliability for performance, we explore techniques that have both low and high reliability impact. Based on the performance/reliability requirements of a system, these techniques can be used individually or in appropriate combinations.

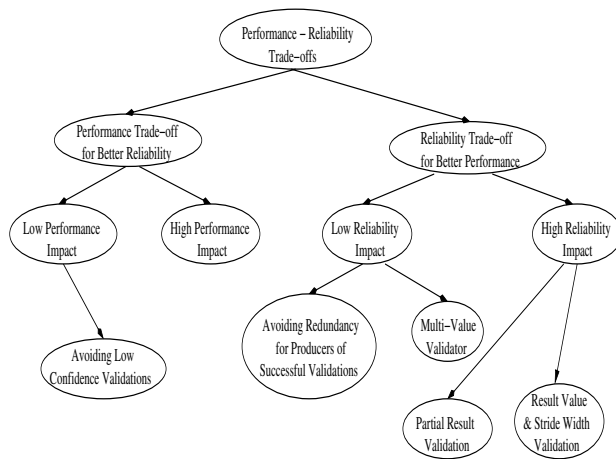


Figure 5. Different Techniques to Explore the Performance-Reliability Trade-off Design Space

## 5.1 Performance Trade-off For Better Reliability – Low Performance Impact

**Avoiding Low Confidence Validations:** Table 6 shows the distribution of the total undetected errors, resulting from main thread instructions, among control and non-control instructions in columns 2 and 6. It shows that about 45% of undetected errors are in the control instructions and about 55% in non-control instructions. The numbers in parenthesis show the percentage of undetected errors within the two categories. Overall, only 0.5% of errors in non-control instructions go undetected, whereas 2.9% of errors in control instructions go undetected. For non-control instructions, the table further presents the percentage of undetected errors due to successful validations in entries without a stride (stride-valid bit is '0'), successful validations in entries with a stride of zero, and successful validations in entries with a non-zero stride. Table 6 shows that a large percentage of undetected errors in non-control instructions are due to successful validations in entries without a stride. The false negative rate in non-control instructions can thus be considerably reduced by disallowing validations for entries with a '0' stride-valid bit. Our experiments showed that by doing just that, the average percentage of total undetected errors (last column in Table 4) reduces from about 0.45% to about 0.23%, with almost no loss in performance. This optimization has negligible additional hardware impact.

For control instructions, Table 6 shows that a majority of undetected errors are incurred in high confidence branch predictions. Confidence in a branch prediction is measured using a saturating confidence counter that is incremented on each correct prediction and decremented on each misprediction. A high confidence prediction is indicated by a confidence counter value greater than a confidence threshold. We experiment with three configurations (i) two-bit counters that are incremented and decremented by one and a confidence threshold of one, (ii) five-bit counters that are incremented and decremented by one and a confidence threshold of 15,

	Branch Mispred. / Correct Result / Detected by Validator					
	Arch. Source	Phys. Source	Operand Value	Result Value	Arch. Dest	Total
vpr	30 / 5 / 64	22 / 7 / 70	15 / 6 / 79	8 / 0 / 92	24 / 4 / 71	21 / 4 / 72
gcc	9 / 1 / 90	5 / 2 / 93	5 / 1 / 94	1 / 0 / 99	13 / 3 / 84	7 / 1 / 92
mcf	34 / 5 / 60	28 / 4 / 67	15 / 4 / 80	6 / 0 / 94	38 / 2 / 58	26 / 3 / 69
parser	40 / 4 / 55	28 / 4 / 65	22 / 7 / 70	13 / 0 / 87	35 / 3 / 60	30 / 4 / 63
vortex	14 / 3 / 82	7 / 4 / 88	8 / 2 / 90	2 / 0 / 98	13 / 8 / 79	10 / 3 / 86
bzip2	17 / 6 / 76	11 / 7 / 80	11 / 13 / 75	4 / 0 / 96	15 / 3 / 81	13 / 6 / 80
wupwise	0 / 2 / 98	0 / 0 / 100	0 / 6 / 94	0 / 0 / 100	0 / 2 / 98	0 / 2 / 98
swim	3 / 9 / 87	2 / 9 / 88	1 / 13 / 86	1 / 0 / 99	3 / 17 / 79	2 / 9 / 88
mgrid	0 / 3 / 96	0 / 2 / 98	0 / 2 / 98	0 / 0 / 100	0 / 2 / 97	0 / 2 / 98
applu	1 / 2 / 97	0 / 2 / 97	0 / 9 / 91	0 / 0 / 100	0 / 3 / 97	0 / 3 / 96
mesa	43 / 7 / 49	33 / 6 / 60	30 / 14 / 56	27 / 0 / 73	38 / 7 / 54	36 / 7 / 51
art	7 / 9 / 83	3 / 11 / 86	4 / 10 / 86	2 / 0 / 98	10 / 4 / 85	5 / 7 / 87
quake	21 / 9 / 68	12 / 10 / 74	7 / 7 / 85	3 / 0 / 96	17 / 5 / 76	13 / 6 / 77
ampp	4 / 5 / 90	5 / 10 / 84	2 / 5 / 93	1 / 0 / 99	5 / 7 / 88	3 / 5 / 92
apsi	14 / 9 / 77	6 / 9 / 84	6 / 9 / 85	3 / 0 / 97	10 / 6 / 83	9 / 7 / 83
Average	16 / 5 / 78	11 / 6 / 82	8 / 7 / 84	5 / 0 / 95	15 / 5 / 79	12 / 5 / 82

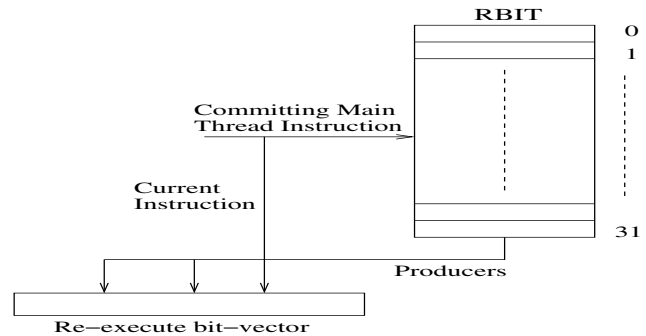
**Table 5. Percentages of errors (out of those that propagate to main thread instructions) that are either masked or detected**

and (iii) five-bit counters that are incremented by one and decremented by three and a confidence threshold of 15. Table 6 shows the results for the 2-bit counter; the results for others were very similar. Redundantly executing control instructions with a low confidence in successful validation only slightly reduces the average total undetected error rate in Table 4 to 0.42%, while the average reduction in the performance impact of SRT reduces to about 52% when conserving the fetch to decode bandwidth. Henceforth, we do not use confidence in control instructions for better reliability.

## 5.2 Reliability Trade-off For Better Performance – Low Reliability Impact

**Avoiding Redundancy for Producers of Successfully Validated Instructions:** The SpecIV technique has an extremely low undetected error rate because it is highly unlikely that a faulty instruction generates an expected result. Similarly, it is highly unlikely that if an instruction is successfully validated, then the instructions producing its operands are executed incorrectly. For instance, we observe in SpecIV that errors in architectural destination register identifiers almost always lead to failed validations of their dependants. Hence, we extend SpecIV such that if an instruction is successfully validated, then even the instructions producing its operands are not redundantly executed. To implement this technique, we provide a *Re-execute Bit-vector Index Table (RBIT)* in the commit stage. RBIT consists of one entry for each architectural register. Each entry stores the re-execute bit-vector index of the instruction producing that architectural register. Each main thread instruction updates the RBIT entry of its destination at commit. This optimization does not require any additional hardware. If a main thread instruction is successfully validated, then in parallel to resetting its re-execute bit, it also resets the re-execute bits at the indices present in the RBIT entries of its source architectural registers. In this technique, all main thread instructions write their results in the CVQ (and their destination register identi-

fiers in DIQ when conserving the fetch to decode bandwidth) because even if instructions are not successfully validated, their redundant counterparts may not be executed because their dependants are successfully validated. This technique increases the processor vulnerability by removing more instructions from the sphere of replication.



**Figure 6. Avoiding Redundancy for Producers of Successfully Validated Instructions**

Our experiments show that this technique avoids redundant execution of an average of about 69% instructions. This results in about 58% performance impact reduction, about 62% when conserving fetch to decode bandwidth, while increasing the undetected error rate from about 0.45% to about 0.5%. However, if the reliability improving techniques of Section 5.1 are combined with this technique, we observed an average undetected instruction error rate of only about 0.09% with an average of about 60% performance impact reduction in SRT.

**SpecIV With Multi-Value Validator (SpecIV-MV):** One method to increase the validation success rate, so as to further reduce the performance impact, is to maintain a larger set of expected values. An instruction's validation succeeds if its result matches any of the expected values. The chances of an

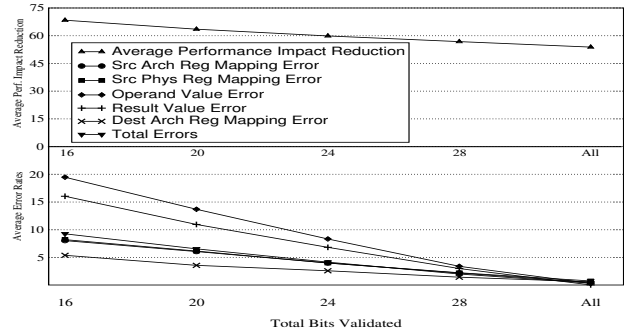
	Non-control Instructions				Control Instructions		
	Out of Total (Within Category)	No stride	Zero stride	Non-zero stride	Out of Total (Within Category)	Low Conf.	High Conf.
vpr	46 (0.6)	82	18	0	54 (3.5)	31	69
gcc	13 (0.0)	89	11	0	87 (2.7)	27	73
mcf	48 (1.0)	94	5	1	4 (2.3)	17	83
parser	27 (0.3)	95	4	1	73 (3.9)	18	82
vortex	13 (0.1)	98	2	0	87 (2.5)	42	58
bzip2	53 (0.7)	88	9	3	47 (3.1)	28	72
wupwise	100 (0.0)	100	0	0	0 (0.0)	0	10
swim	94 (0.5)	77	20	3	6 (0.2)	18	82
mgrid	88 (0.0)	44	34	22	12 (0.3)	5	95
applu	57 (0.2)	78	22	0	43 (11.9)	0	100
mesa	65 (0.3)	95	4	1	35 (0.6)	39	61
art	63 (0.3)	39	58	3	37 (1.9)	16	84
equake	77 (2.2)	79	11	10	23 (5.7)	4	96
ammp	41 (0.1)	87	12	1	59 (0.7)	7	93
apsi	44 (0.3)	89	9	2	56 (3.5)	38	62
Average	55 (0.5)	82	15	3	45 (2.9)	19	81

**Table 6. Confidence in successful validations for undetected errors from main thread instructions**

instruction’s incorrect result matching an expected value also increases, thus reducing reliability. In SpecIV-MV, we maintain an additional direct mapped 1K-entry validator which records the last four distinct values in each entry for those instructions whose validations fail in the main validator. We observed that SpecIV-MV was not very effective as compared to SpecIV because it further reduces the performance impact of SRT by only about 2%, while increasing the average undetected errors from about 0.45% to about 0.91%. We also experimented with storing last four strides instead of values, but it performed worse than SpecIV-MV.

### 5.3 Reliability Trade-off For Better Performance – High Reliability Impact

**Partial Result Validation:** Another method to relax the conditions for successful validations is to validate only a subset of result bits. This approach increases vulnerability because of the inability to detect errors in the ignored bits and can have a high reliability impact depending on the number of bits ignored. Figure 7 plots the average percentage performance impact reduction and the average undetected instruction error rate when validating only the least significant 16, 20, 24, and 28 result bits and also when validating all the bits for the base SpecIV. A 64-bit result is treated as two 32-bit results. The average performance impact reduction increases from about 54% to about 69% when the validated bits are reduced from 32 to 16, and the average total undetected instruction error rate increases from about 0.45% to about 10%. When validating fewer bits, the machine becomes more susceptible to errors in operand and result values because errors in them directly correspond to errors in the results. For instance, errors in the higher order bits of operands lead to errors in the higher order bits of results, which may be ignored in this scheme resulting in undetected errors. The errors in register identifiers, on the other hand, impact random bits in the results.

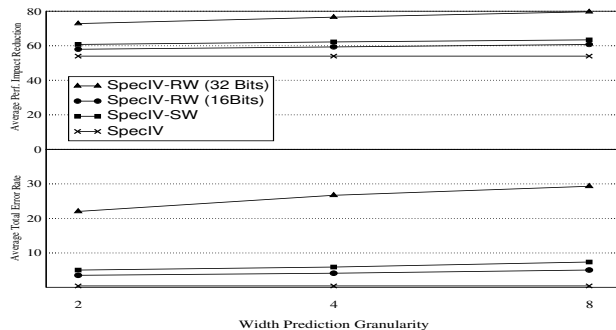


**Figure 7. Undetected Instruction Error Rate and Performance Impact Reduction of Partial Result Validation**

**Result Value Width and Stride Width Validation:** In these techniques, an instruction is not re-executed if its result’s width matches the expected width (SpecIV-RW) or the width of its result’s difference from the last result matches the expected width (SpecIV-SW). These validations are performed for only those instructions whose result value validations fail. Each validator entry is augmented to store either the last result’s width for that entry in SpecIV-RW or the width of the difference in the two most recent result values (Stride Width) for that entry in SpecIV-SW. Width of a value is determined by the position of the most significant ‘1’ in its representation. We also define width granularity that specifies the acceptable variations in width. For instance, if the width of an expected value is 4, then values with widths 3, 4, and 5 result in successful validations for a width granularity of 2, and values with width 2,3,4,5, and 6 result in successful validations for a width granularity of 4.

Figure 8 shows the average total undetected instruction error rates and average percentage reduction in performance impact of SpecIV-RW and SpecIV-SW for width granularities of 2, 4 and 8, on top of the base SpecIV. For SpecIV-RW,

we also experiment with limiting the result width validations for only those results that require less than or equal to 16 bits for representation. Figure 8 shows that both SpecIV-RW and SpecIV-SW have a high reliability impact. However, they also significantly reduce the performance impact, as high as 80% for SpecIV-RW with 32 bits. Performance impact reduction and average undetected errors also increase with increasing granularities.



**Figure 8. Performance impact reduction and undetected instruction error rates for SpecIV-RW and SpecIV-SW**

## 6 Conclusion

Reliability in RMT systems is ensured by corroborating the results of redundant threads. Redundant thread execution has a significant impact on the performance of the processor. For contemporary systems, achieving perfect reliability at the expense of considerable performance impact is not a good design option because the error rate achieved with a slightly vulnerable system may still fall within the target error rates for the system.

In order to achieve a good performance-reliability trade-off, we explore a simple to implement speculative mechanism that validates the execution of an instruction by comparing its result against the expected value. Only those instructions are redundantly executed for which the validations fail. This approach, along with a combination of several performance-reliability trade-off techniques discussed in the paper, incurs a performance impact of only about 10% while incurring an average undetected instruction error rate of only about 0.09%, as compared to SRT model that incurs a performance impact of 25% with zero vulnerability. Our approach also significantly outperforms other techniques to reduce redundancy in SRT. We also discuss several techniques that further trade-off large amounts of reliability for significantly better performance.

## References

- [1] T. Austin, "DIVA: a reliable substrate for deep sub-micron microarchitecture design," *Proc. Micro-32*, 1999.
- [2] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Arch. News*, 1997.
- [3] J.A. Butts and G. Sohi, "Dynamic dead instruction detection and elimination," *ASPLOS*, 2002.
- [4] Compaq Computer Corp., "Data integrity for Compaq Non-Stop Himalaya servers," <http://nonstop.compaq.com>, 1999.
- [5] Martin Dimitrov and Huiyang Zhou "Locality-Based Information Redundancy for Processor Reliability," *WAR*, 2006.
- [6] M. Gomaa, et. al., "Transient-Fault Recovery for Chip Multiprocessors," *Proc. ISCA-30*, 2003.
- [7] M. Gomaa and T. N. Vijaykumar, "Opportunistic Transient-Fault Detection," *Proc. ISCA-32*, 2005.
- [8] M. K. Gowan, et. al., "Power Considerations in the Design of the Alpha 21264 Microprocessor," *Proc. DAC*, 1998.
- [9] J. G. Holm, et. al., "Low cost concurrent error detection in a VLIW architecture using replicated instructions' *Proc. ICPP-21*, 1992.
- [10] S. Kumar and A. Aggarwal, "Self-Checking Instructions: Reducing Instruction Redundancy for Concurrent Error Detection," *Proc. PACT*, 2006.
- [11] K. Lepak and M. Lipasti, "Temporally silent stores," *Proc. ASPLOS*, 2002.
- [12] Jose F. Martinez, et. al, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors," *Proc. Micro-35*, 2002.
- [13] S. Mukherjee, et. al., "A Systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," *Micro-36*, 2003.
- [14] A. Parashar, et. al., "SlicK: slice-based locality exploitation for efficient redundant multithreading," *Proc. ASPLOS*, 2006.
- [15] A. Parashar, et. al., "A Complexity-Effective Approach to ALU Bandwidth Enhancement or Instruction-Level Temporal Redundancy," *Proc. ISCA*, 2004.
- [16] J. H. Patel, and L. T. Fung, "Concurrent error detection in ALU's by recomputing with shifted operands," *IEEE Transactions on Computers*, 31(7):589-595, July 1982.
- [17] P. Racunas, et. al., "Perturbation-based Fault Screening," *Proc. HPCA-13*, 2007.
- [18] J. Ray, et. al., "Dual use of superscalar datapath for transient-fault detection and recovery," *Proc. Micro-34*, 2001.
- [19] V.K. Reddy, et. al., "Understanding Prediction-Based Partial Redundant Threading for Low-Overhead, High-Coverage Fault Tolerance," *Proc. ASPLOS XII*, 2006.
- [20] S. Reinhardt, and S. Mukherjee, "Transient fault detection via simultaneous multithreading," *Proc. ISCA-27*, June 2000.
- [21] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," *Proc. of the 29th Intl. Symp. on Fault-Tolerant Computing Systems*, June 1999.
- [22] D. P. Siewiorek and R. S. Swarz, "Reliable Computer Systems Design and Evaluation," *The Digital Press*, 1992.
- [23] T. J. Slegel, et al. "IBM's S/390 G5 microprocessor design," *IEEE Micro*, 19(2):12-23, March/April 1999.
- [24] J. Smolens, et. al., "Efficient Resource sharing in Concurrent error detecting Superscalar microarchitectures," *Proc. Micro-37*, 2004.
- [25] A. Sodani and G. Sohi, "Dynamic instruction reuse," *Proc. ISCA*, 1997.
- [26] N. Soundarajan, et. al., "Mechanisms for Bounding Vulnerabilities of Processor Structures," *Proc. ISCA*, 2007.
- [27] K. Sundaramoorthy, et. al., "Slipstream processors: Improving both performance and fault tolerance," *In Proc. Micro-33*, December 2000.
- [28] T. Vijaykumar, et. al., "Transient-fault recovery using simultaneous multithreading," *Proc. ISCA-29*, 2002.
- [29] K.R. Walcott, et. al., "Dynamic Prediction of Architectural Vulnerability from Microarchitectural State," *Proc. ISCA*, 2007.
- [30] N. Wang and S. Patel, "ReStore: Symptom Based Soft Error Detection in Microprocessors," *Proc. DSN*, 2005.
- [31] C. Weaver, et. al., "Techniques to Reduce the Soft Error Rate of a High Performance Microprocessor," *Proc. ISCA-31*, 2004.