End-to-End Register Data-Flow Continuous Self-Test

Javier Carretero, Pedro Chaparro, Xavier Vera, Jaume Abella, Antonio González
Intel Barcelona Research Center, Intel Labs – UPC
Barcelona, Spain
{javierx.carretero.casado,pedro.chaparro.monferrer,
xavier.vera, jaume.abella,antonio.gonzalez}@intel.com

ABSTRACT

While Moore's Law predicts the ability of semi-conductor industry to engineer smaller and more efficient transistors and circuits, there are serious issues not contemplated in that law. One concern is the verification effort of modern computing systems, which has grown to dominate the cost of system design. On the other hand, technology scaling leads to burn-in phase out. As a result, in-the-field error rate may increase due to both actual errors and latent defects. Whereas data can be protected with arithmetic codes (like parity or ECC), there is a lack of cost-effective mechanisms for control logic.

This paper presents a light-weight microarchitectural mechanism that ensures that data consumed through registers are correct. Microarchitecture presents a new way to manage reliability and testing without significantly sacrificing cost and performance, offering a unique opportunity to detect errors in the field at low cost. Our results show a coverage around 90% for the targeted structures with a cost in power and area of about 4%. The structures protected include the issue queue logic and the data associated (i.e., tags, control signals), input multiplexors, rename data, replay logic, register free list, bypasses data and logic, MOB data and addresses, register file logic, register file storage and functional units.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Reliability, Testing and Fault-Tolerance; C.4 [**Performance of Systems**]: Reliability, Availability and Serviceability of Systems

General Terms

Design, Performance, Reliability

Keywords

Online testing, end-to-end protection, degradation, design errors, control logic

ISCA'09, June 20–24, 2009, Austin, Texas, USA.

1. INTRODUCTION

The reliability, availability and serviceability (RAS) of systems to perform to customer expectations are strongly related to how the system is designed to respond to both hard and soft failures. Different sources of errors challenge the RAS of current designs: inherent design errors, hard errors, degradation and transient errors.

Design and hard errors are usually detected at design testing time, which is the process of detecting bugs by running the processor with random and special inputs. There is an increasing recognition that validation efforts have difficulties in keeping pace with the growing processor design complexity. It is very complex to test a system with all possible inputs, and there will always be hard-to-test corner cases. Therefore, some hard errors may slip through the scanning phase and may show up during operation.

In addition, burn-in is becoming less effective: thermal runaway due to high leakage and longer test patterns due to the increased gate count diminish the effectiveness of burn-in [12]. As a consequence, small latent defects may become large enough during operation due to degradation and cause failures before the target lifetime [6, 10].

Finally, transient errors are an important challenge in modern microprocessors. Error sources such as soft errors, erratic bits or variations also threaten technology scaling and require detection and correction mechanisms in place [2].

All such faults may translate into frequent errors during operation that must be detected and corrected timely to allow fault-free operation. Applying traditional fault tolerance techniques, such as error correcting codes, redundancy, re-execution or adapting off-line testing would be a straightforward solution. However, the non-negligible costs that traditional fault tolerance techniques have and the important cost constraints that generally characterize consumer electronics make them too expensive.

Critical components from the point of view of their susceptibility to faults, such as large RAM and SRAM structures and datapaths, are already protected with parity or ECC in many commercial processors [3, 20, 23]. However, while arithmetic codes could be used to protect the information stored in some structures like the issue queue (e.g., opcode, operand values, operand tags, etc.), the correct operation of the logic cannot be guaranteed (e.g., issue selection logic, etc.). Limited research efforts have been devoted to cost-effective error detection strategies for the control logic of high-performance microprocessors, even though it plays a critical role for the whole microprocessor correct operation, and it represents a significant portion of the die area.

End-to-end protection is an efficient way to achieve wide coverage at a small cost. The concept of end-to-end protection is based on identifying a path either for data or instructions where there is a source from which data or instructions originate, and a consumption site where they are finally consumed. The end-to-end scheme involves generating a protection code at the source, sending the data or instructions with the protection code along the path, and checking for errors *only* at the end of the path, where data or instructions are consumed. Any error found at the consumption site can be caused by any logic gates, storage elements, or busses along the path.

This paper proposes a low-cost *online end-to-end* protection scheme that verifies the register data flow. The proposed end-to-end scheme allows detecting hard errors, errors caused by degradation and intermittent errors such as soft errors. Moreover, the solution is based on microarchitectural invariants, which allows detecting design errors (bugs).

The centerpiece of the proposed method is a *signature-based* protection mechanism that protects the control logic involved in the register data flow. This includes the *rename* tables, *wake-up* logic, *select* logic, *input multiplexors*, *operand read/writeback*, the *register free list* data and logic, and the *replay* logic. An important feature of our mechanism is that it can work standalone or it can be integrated smoothly with other end-to-end error detection techniques. We describe how we can integrate our data-flow end-to-end protection mechanism with an end-to-end residue checking approach that protects the *functional units*, *MOB* data and addresses, and the *register file* storage without additional cost.

The rest of the paper is structured as follows: Section 2 reviews some concepts that will be used throughout the paper and defines the baseline core. Section 3 explains our proposal for a data-flow self-test mechanism. Section 4 details how to extend it to the replay logic. Finally, Section 5 overviews an end-to-end residue coding scheme and explains how to integrate it with our proposal. Section 6 evaluates coverage and overheads, while Section 7 reviews some relevant related work. We summarize our main conclusions in Section 8.

2. OVERVIEW

In this section, we first introduce a modern processor design and assess which are the most commonly protected components. Then, we identify how faults in the unprotected structures that control the data flow may impact the execution of programs, and finally, we give an overview of our end-to-end proposal.

2.1 Baseline Core

Figure 1 shows a high-level description of our baseline microarchitecture. It resembles an Intel® Core™ Micro-Architecture design except for the fact that it implements a physical register file (RF) [9]. Intel® 64 *macro-instructions* are translated into *micro-instructions* (*micro-ops*). Without loss of generality, we assume *micro-ops* consume two sources and generate one value.

2.2 RAS Features in the Baseline Core

The baseline RAS features in most commercial processors usually consist of the use of parity and ECC codes for arrays, parity for datapaths, and the use of hardened latches. Figure 1 shows in grey the arrays that we assume protected by an error code scheme. Usually, large arrays like second level caches are protected with ECC, while smaller ones like first level caches or register files employ parity. We also assume that the control logic of cache-like structures is protected with existing online techniques [1, 27]. We do not use such techniques for protecting the register file logic because the overhead is too large.

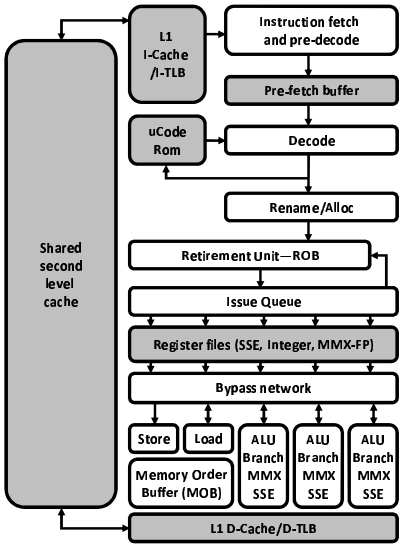

Figure 1: Baseline microarchitecture. We show in grey those blocks that are protected by existing techniques

Figure 1 shows in white boxes the amount of logic that cannot be protected by existing mechanisms. Some of the challenges include the rename logic, the issue queue, bypass network, ROB and the free list, which have a critical role in enforcing a correct data flow. Notice that although most of the commercial processors do not protect the functional units, arithmetic codes [5, 13] such as residue coding, could be used to protect them.

Faults in the data flow could result into different architectural errors. We classify them by error location, and depict which possible faults caused them.

1. *Selection of wrong inputs*: The input multiplexors that choose the input operands from the bypass/RF may select a wrong input, causing an incorrect data to be consumed.

2. *Wrong register file access*: A read access to the register file may provide a wrong data value. The causes might be: (a) a "register read" reads from a wrong entry, and (b) a "register write" writes into a wrong register; the readers will suffer the consequences.

3. *Premature issue*: A prematurely issued instruction will consume a wrong data value. The ultimate causes include: (a) incorrect operation of the wake-up logic, (b) incorrect operation of the select logic, and (c) incorrect assignment of the latency of a producer instruction (the consumers suffer the effects).

4. *Wrong tag*: An instruction may depend on a wrong

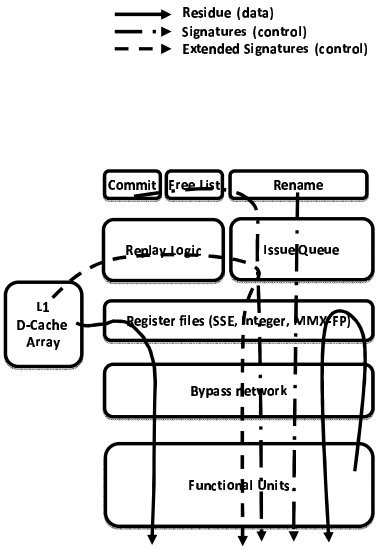

Figure 2: End-to-end paths

instruction (i.e., through a wrong *tag*) and consume its data. The causes might be: (a) faults in the rename dependence checking inside the rename bundle, (b) incorrect contents in the rename table, (c) wrong access to the rename table, and (d) corruption of a *tag* tracking a register dependence.

5. *Data stall in the bypass network*: If the latches placed in the different levels of the bypass do not latch a new value (i.e., due to a missing or delayed clock signal) it may happen that it gets stalled with an old data value.

6. *Register free-list misuse*: If the register free list does not operate correctly (including wrong register release and allocation), the *tags* might get corrupted. We also consider the situation when the old or current mapping in the ROB may get corrupted. This may cause physical registers to be simultaneously the destination location for two different instructions.

7. *Load replay errors*: If the replay logic does not work properly, it may neither identify nor reissue all the instructions that depend on a load that misses in the data cache. As a consequence, there could be silent commitment of bogus values, potentially corrupting the architectural state.

8. *Deadlock*: A deadlock will happen if the oldest instruction waits (incorrectly) for a *tag* that is not in-flight and, hence, cannot trigger a wake-up. This is a subcase of a "wrong tag" with a different microarchitectural result.

Faults that result in a deadlock can be easily detected by means of a watchdog timer, already implemented in many current microprocessors. However, the other faults result in instructions operating with a wrong data value, and are the target of our protection mechanism.

2.3 Overview of the End-To-End Proposal

Figure 2 shows the end-to-end protection scheme. As one can see, there are different end-to-end paths. We summarize

Structure	Signatures E2E	Residue E2E
Issue Queue	wake-up logic (3) select logic (3) tags (4 & 8)	immediate field
Rename	rename table (4b,d)	
Bypass	logic (5)	data
Replay System	replay logic (7)	
Free List	tags (6) register release (6) register allocate (6)	
ROB	old map (6) current map (6)	
MOB		data & addresses
Register File	access (2)	data
FUs	wrong inputs (1)	wrong operation

Table 1: Protected structures and their corresponding error location (between parenthesis)

in Table 1 how our implementation of each end-to-end path contributes to protect different structures. The table also relates every structure to the corresponding error location described above (see Section 2.2). One of the end-to-end paths covers the data flow starting from the rename table, going through the control of the issue queue (dashed-dotted lines). Another path (dashed lines) covers the replay logic. Finally, few other end-to-end paths cover the data (bold lines). Notice that cases 4(a) and 4(c) that relate to faults in the rename logic are not covered by the presented implementation and are left as future work.

In the next sections, we will explain how each kind of end-to-end path is implemented. We will also show how they are integrated sharing the same hardware infrastructure, and therefore, avoiding unnecessary overheads.

3. END-TO-END DATA-FLOW TESTING

This section describes our proposal for an efficient mechanism to perform online testing of the register data flow executed by the processor (dashed-dotted path in Figure 2).

3.1 Overview

We propose a novel technique that is based on marking every data value flowing through the pipeline with a *signature*. Whereas codes such as residue, parity or ECC are a function of the data they are associated with, signatures in its general definition do *not* depend on any property of the protected information.

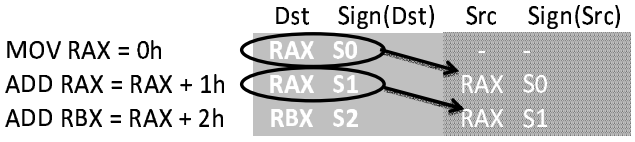

Figure 3: Signature assignment among dependent instructions

Our proposed online testing technique is exemplified in Figure 3, which shows three instructions with their corresponding destination and source signatures. Each operand, including sources and destination, receives a signature that allows tracking the data flow. Each *a priori* source signature is compared with *a posteriori* signature obtained during ex-

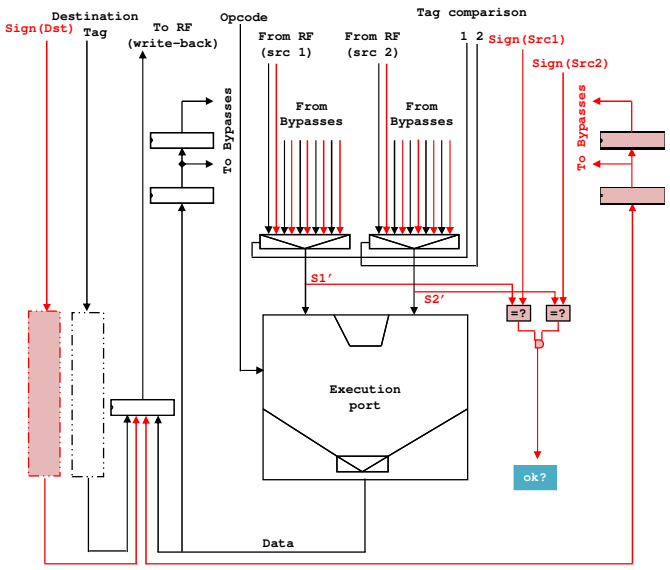

Figure 4: Backend modifications. Signatures are shown in grey

ecution. If both signatures mismatch, an error is detected. Otherwise, the destination signature is written back and forwarded to any potential consumer along the data. In this way, the producer-consumer loop is continuously monitored.

3.2 Signature-Based Protection

We describe the signature-based protection (see Figure 4) dividing it into three steps: signature assignment, signature flow and signature check. We will also describe the changes in the processor stages and hardware structures required for implementing the proposed mechanism.

3.2.1 *Signature Assignment*

Signature assignment is performed in two steps, similar to register allocation. Those instructions that generate a value obtain a destination signature at rename time. We take as a baseline policy to generate the signatures a simple auto-incremented counter (*round-robin* policy). We will discuss in Section 6 its complexity and coverage.

Source operands receive their corresponding signatures also at the rename stage. In a fault-free scenario, such signatures match the destination signature of the producer instruction of that operand. Overall, every instruction flows after the rename stage with 3 signatures: the signature of the data it will produce ($Sign(Dst)$), and the signatures of the producers of its two operands ($Sign(Src_1)$ and $Sign(Src_2)$).

Immediate values are also assigned a signature. Since the number of immediate operands is low, we will concentrate in the rest of the paper on register operands.

3.2.2 *Signature Flow*

Figure 5 depicts how data and signatures flow through the pipeline. The issue queue receives instructions with two source and one destination signatures. At the issue queue, instructions wait until they are ready for execution. Once the instruction issues, it reaches the multiplexors at the input of the functional units. Such multiplexors select among the different data sources (i.e., the available bypasses and register file) the value to consume. Therefore, since data and signatures travel together, the operand selection also selects the signature that flows with it.

Once the instruction finalizes execution, it sends the result with its corresponding signature ($Sign(Dst)$) to both the bypass network and the register file. By sending the signature through the bypass and storing it in the register file, subsequent instructions that consume it (either correctly or wrongly) can perform the flow check.

3.2.3 *Signature Check*

The data-flow checks are performed after the input multiplexors select the data that the functional unit will use. At that point, we have all the information required to validate that everything went right: (i) the signatures of the data values that the functional unit is going to use, and (ii) the signatures of the values the functional unit *should* consume, which were obtained at the rename stage. Two signature comparators are placed next to the functional units; if any of the signatures mismatches, an error is detected.

3.2.4 *Hardware Modifications*

Figure 4 shows a close view of the hardware changes required in the backend. Notice that since all logic executes in parallel with existing hardware, this approach can detect design errors. A detailed list of the microarchitectural changes follows (assuming B bits per signature for a total of $M = 2^B$ signatures).

- **Rename stages.**
  - A counter of B bits to implement the baseline *round-robin* policy.
  - An additional B-bit field per register in the rename table to store the last assigned signature.
  - Wider multiplexors to select the proper signature for the operands of every instruction.
  - Additional fields in the output latches to carry the signature information (3 signatures of B bits).
- **Issue queue.** The CAM memory is left unchanged. Conversely, we enlarge the RAM. Each entry in the payload RAM will hold extra fields for keeping the signatures of the sources and the destination (3 signatures of B bits). Input write ports and output read ports are resized accordingly.
- **Register files.** Additional space and wires to store the signature per register (B bits per register).
- **Bypass network.** Additional wires to carry the signature of each value (B bits per value), and wider input multiplexors.
- **Execution units.** Signature checkers that compare the signature of the value received in execution units with the expected signature (2 comparators of B bits).
- **Write-back network.** Additional wires to carry the signature of each value (B bits per value).

Large part of the hardware overhead mainly comes from (i) the additional register file storage, (ii) the additional

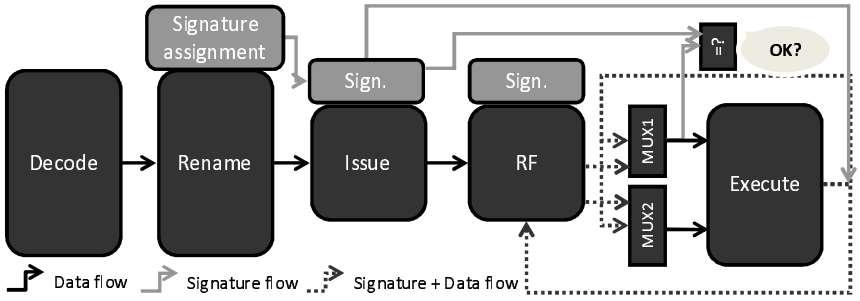

Figure 5: Signature and data flow

wires in the bypass network, and (iii) the additional storage in the issue queue. Therefore, the signatures should be as narrow as possible. However, reducing the signature width decreases the total number of available signatures. In that case, it may happen that in the presence of a fault, an incorrect operand turns out to have the same signature as the expected one (i.e., *aliasing*). We discuss these trade-offs in Section 6.

4. EXTENDING PROTECTION TO LOAD REPLAY

In this section we explain how we can extend the end-to-end data-flow protection mechanism proposed in the previous section to perform online testing of the replay logic (dashed end-to-end path in Figure 2).

Load hit speculation predicts the access time of load instructions in order to allow instructions dependent on a load to issue back-to-back, without having to wait for actual hit status. A misprediction implies that correct instructions consume the wrong data. Therefore, those instructions consuming wrong data are re-issued later with correct data. This mechanism is called *load replay system*.

Our technique to protect the replay mechanism is built on top of the end-to-end data-flow explained in Section 3: a small circuit (*spoil circuit*) corrupts the destination signature for each instruction in the shadow of a missing load, in such a way that dependent instructions will detect an error through the regular signature checking.

The corruption process starts with the speculative load missing in cache. Once the latency misprediction is detected, the destination signature of the load is corrupted. At the execution unit, the direct dependent instructions that need to be replayed corrupt their own destination signature; therefore, all consumers that depend on the load latency misprediction, will receive a corrupted signature.

Signatures are checked regularly at the execution units by comparing the received signature and the expected signature. In parallel, we use the information provided by the replay logic, which tells whether an instruction should be replayed or make forward progress. Notice that in case of a load latency misprediction, the logic expects a mismatch in the signatures. Depending on the outcome of the replay logic and the signature checks, our technique performs different actions:

- If both agree, it indicates a correct execution, and therefore, nothing needs to be done.
- If both outcomes disagree, it indicates that something went wrong in the replay logic. Then, an error signal is triggered.

4.1 Hardware Modifications

The added hardware to implement the error detection for the replay logic is just the *spoil circuits*. The *spoil circuits* can be implemented with just an inverter, since we only want to corrupt the signatures. We require one *spoil circuit* for every functional unit that propagates a destination register and signature.

5. END-TO-END DATA ONLINE TESTING

This section starts with an overview of residue coding. Next, we outline a possible end-to-end implementation that protects the data based on residue coding. Finally, we detail how to integrate it with our end-to-end data-flow protection mechanism, in such a way that their overheads are shared.

5.1 Residue Coding Overview

Arithmetic codes have been deeply studied in the past for protecting both the data and arithmetic and logic functional units. They are based on attaching a redundant code to every data word. While data is protected by verifying the redundant code, arithmetic operations are protected by operating in parallel the data and the codes.

We decide to use residue codes [5, 13]. In the past, it has been shown to protect arithmetic operations [5, 13, 22, 37] and logical operations [19, 33]. Recently, it has been extended to floating point functional units [14, 28].

Residue coding computes the redundant code as the modulo N representation of the data value being protected

$$R(x) = x \mod N$$

We choose $N = 3$ (we need two bits to represent the residue of any number); previous works [14, 22, 37] show that coverage is high (about 90%) with a small cost (it will be discussed in Section 6).

5.2 Implementing an End-to-End Residue

If one only focuses in protecting the functional units, the cheapest solution in terms of hardware is pre-computing the residue codes right before feeding the functional units. However, we want to protect both the data path and the functional units (bold end-to-end paths in Figure 2).

Figure 6 shows an implementation of such end-to-end residue coding. Residue codes are calculated where data is originated: (i) loads from the L1 data cache, and (ii) output from

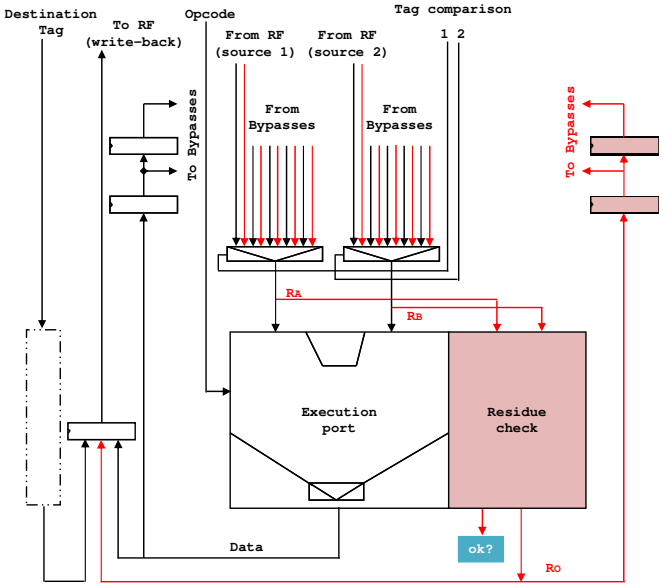

Figure 6: End-to-end residue coding. Residue hardware is shown in grey

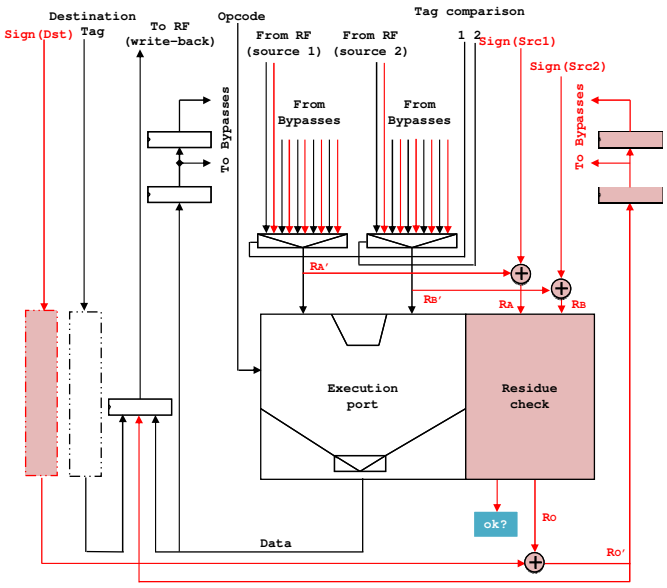

Figure 7: Scheme for a combined protection technique with signature protection and residue coding. We show in grey the hardware extensions

the functional units. Residue codes flow through the bypass network, and are stored in the register file. In this way, data is protected in an end-to-end fashion: from the point it is originated, to the point it is consumed. Notice that for this implementation, residue coding substitutes parity in the register file, since both protect the data. Correctness of functional units is achieved by the residue checkers placed next to them.

5.2.1 *Hardware Modifications*

This mechanism requires the following hardware modifications (assuming K bits per residue).

- **L1 data cache.** A residue generator for every read port.
- **Register files.** Additional space to store the residue per register (K bits per register).
- **Bypass network.** Additional wires to carry the residue of each value (K bits per value). In addition, wider input multiplexors to obtain the proper residue per operand are required.
- **Execution units.** A residue unit that operates with the incoming residues, a residue generator for the produced value by the functional unit, and a residue checker that validates both operands. For those units that cannot operate with the incoming residues, we only need the residue checker and a residue generator for the produced value (if any).
- **MOB.** Additional space to store the residue for the data and address per entry.
- **Write-back network.** Additional wires to carry the residue of each value (K bits per value).

5.3 Integrating Signatures with Residues

Comparing Figure 4 and Figure 6, and the detailed hardware modifications listed in Section 3.2.4 and Section 5.2.1, one observes that signatures and residue implementations have pretty much the same hardware requirements. Therefore, we propose to merge the calculated signatures attached to values with the residue values.

In order to use the same hardware to implement both error detection techniques, we encode a new residue value that is a function of the original residue and the signature of the destination register. Similarly, each encoded residue value is decoded back to the original residue using the signature of the corresponding source.

The transformation function must be easy to implement. Besides, it has to be possible to construct the inverse function, so residues can be recovered. An extremely simple and fast function of this type is the bitwise XOR function. Finally, we only leave the residue checkers and remove the signature checker. If there is an error in any of the different end-to-end paths that we protect, we will not be able to decode a correct residue, and the residue checker will suffice to flag the error.

The whole encode/decode process is depicted in Figure 7. When an instruction is executed it writes back `Data` into the register file. Then, its residue `Ro` is XOR-ed with the destination signature `Sign(Dst)` of the instruction writing back. This encoded residue `Ro'` will be written back into the register file and will travel through the bypass network.

Clearly, a consumer instruction requires the correct signature to retrieve the original residue `Ro`. Consumers use the signatures received from the rename stage (`Sign(src1)` for the left operand and `Sign(src2)` for the right one) to decode the input residues (R'_A and R'_B). If an error happens in the data flow, the decode process will generate an incorrect residue. Therefore, when the residue checker operates it will detect the error.

Parameter	Value
Memory	Latency of 45ns
UL2	512 KB, 8-way, 12 cycle hit, 1 R/W port
DL1	32 KB, 4-way, 3 cycle hit 2 read + 1 write port
I-Cache	32 KB, 8-way, 3 cycle hit 1 read + 1 write port
DTLB/ITLB	128 entries, 8-way
ROB/MOB	128 - Commit of 4 / 30 loads, 20 stores
Register File	128 Int, 128 FP, 2 bypass levels
Issue Queue	32 entries scheduler, 6 issue up to 3 Ints, up to 2 FP
Integer Units	3 ALUs, 2 AGU, 1 multiplier
FP Units	1 adder, 1 multiplier, vector

Table 2: Baseline processor

5.3.1 Load Replay Protection

We have explained in Section 4 how to protect the load replay logic by spoiling the signatures. When combining the signatures with the residues, the implementation of the *spoil circuit* also changes. Instead of corrupting the signatures, we corrupt the residues.

When working with modulo $N = 3$, the residue has an invalid value, which is the `all ones` (i.e., only values `00`, `01` and `10` are possible with modulo 3). We spoil residues by using the invalid residue value `11`, in such a way that when output residues are compared, there will always be a mismatch, since residue generators will never generate this value. In order to propagate the corrupted residue until it is consumed, we also need to make sure that the functional units that operate with residues deal with the invalid residue. We modify them in such a way that whenever one of the sources is the invalid residue, the output will be the invalid residue. In this way, all instructions depending on the missing loads will observe a wrong output residue.

6. EVALUATION

This section presents a detailed evaluation of the framework presented. We evaluate it in terms of area, power and coverage for a modern out-of-order processor.

6.1 Methodology

We have conducted experiments with a processor that resembles the Intel® Core™ Micro-Architecture (see Section 2). The particular processor configuration is described in Table 2. Results were collected from an industrial cycle-accurate trace-driven simulator which simulates the functional and microarchitectural behavior of the pipeline. Since coverage of residue coding has been deeply studied in previous works [5, 14, 22, 37] and shown to be about 90% for 2-bit residue codes, we only evaluate in detail its cost. We also evaluate in detail coverage and cost for the data-flow protection mechanism.

Data-flow coverage is assessed by computing the capability of our framework to detect the faulty situations described in Section 2.2: (1) *Selection of wrong inputs*, (2) *Wrong register file access*, (3) *Premature issue*, (4b&4d) *Wrong tag*, (5) *Data stall in the bypass network*, (6) *Free-list misuse*, (7) *Load replay errors*, and (8) *Deadlock*.

Load replay errors has 100% coverage since we enforce the usage of a wrong residue to trigger the error detection mechanism. For the *Deadlock* case, a watchdog time is enough. Therefore, for the rest of the evaluation, we will concentrate on cases (1)–(6).

Since the vulnerability of a processor depends on the dynamic behavior (and thus, the programs run), we opt to run the well known SPEC CPU2000 [38] benchmark suite. Our workload consists of 26 traces of 100 million instructions. For each benchmark, we perform 1082 error injections for each class of error independently. Error injection is performed at the microarchitectural level.

6.2 Coverage Results

Given a tag signature, the probability to match another tag signature will depend on the total number of signatures and the way they are generated/assigned. An error may not be detected if the signature observed when there is an error is the same as the expected one (i.e., *aliasing*). When using B bits to encode the signature, assuming they are uniformly distributed and used, the average case probability of having aliasing is $\frac{1}{2^B}$. Therefore, the expected average-case coverage of our technique in this case would be $1 - \frac{1}{2^B}$. For example, if $B = 2$ then fault coverage would reach 75%, for $B = 3$ it would be 87.5%, whereas if $B = 4$ then fault coverage would be 93.75%. Next, we compare this theoretical numbers with results obtained experimentally. We focus on 2-bit and 3-bit signatures, since the small number of signatures has a larger impact on the expected average-case coverage and the efficiency of the signature allocation policy. We also discuss few optimizations to the baseline *round-robin* policy for improving coverage when the numbers of signatures is small.

Figure 8 depicts the coverage provided by our framework. We include different error sources, assignment policies and signature width. We do not consider a signature assignment policy based on the physical tag because faulty situations like (3) *Premature issue* would remain unprotected. Instead, a random assignment policy, `RAN`, is shown for comparison purposes. We find that coverage is usually better than the expected average case. This indicates that despite signatures are provided in a balanced manner, they are actually used in a way that allows better opportunities for coverage.

We start analyzing the *round-robin* policy (`RR`). We can observe that when we only have 4 signatures, the coverage is below the expected average for the *premature issue* case. However, it works fine when the number of signatures is larger. We have found out that *round-robin* suffers a "wrap-up phenomenon". This is caused by: (i) the *round-robin* policy is implemented as a modulo counter, (ii) the number of registers (128) is multiple of the number of signatures, and (iii) there is a high probability of repeating a signature for the same physical register because the physical registers tend to be used round-robin.

In order to mitigate this problem, we have experimented with a *pseudo round-robin* policy. Instead of having just one modulo counter, we use multiple of them. Then, each logical register is statically paired to one of the counters. In this way, we avoid the signature wrapping while maintaining the benefits of an homogeneous signature distribution. We named these pseudo round-robin schemes `DRR` when using 2 counters, and `QRR` when using 4 counters. Our results show that for 2-bit signatures `DRR` does not help since we do

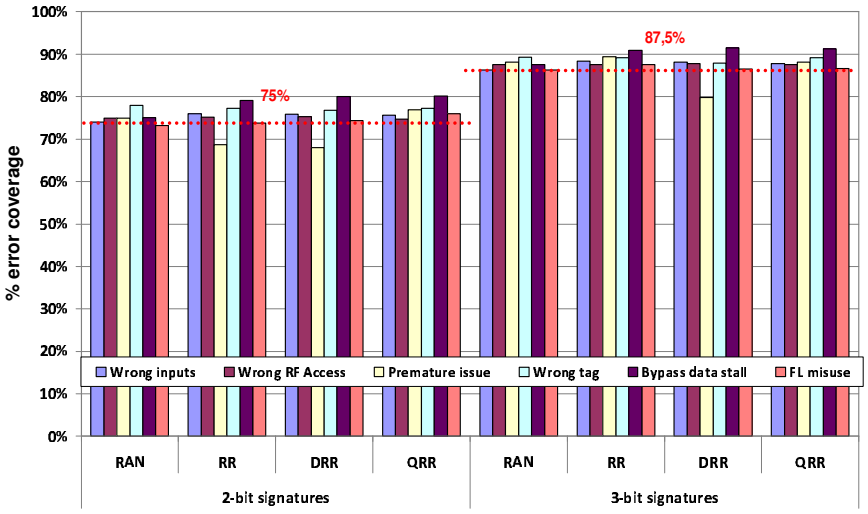

Figure 8: Coverage for different assignment policies and different errors

not increase enough the signature variability. However, QRR increases coverage from 69% up to 76%. For 3-bit signatures, we have observed that all round-robin policies behave very similarly, since in that case the variability is high enough to naturally avoid the wrap-up phenomenon.

When comparing the RAN against *round-robin*, we notice that although they have very similar coverage, errors like *data stall in the bypass network* benefit a lot from the round-robin distribution. The reason behind is that *round-robin* policy maximizes the distance between two consecutive uses of the same signature, and therefore, the probability of reading a stalled latch with the same expected signature is lower than when using a random policy.

In conclusion, we can see that simple allocation policies allows us obtaining the expected theoretical coverage with low cost.

6.3 Overheads

This section details the impact of our framework in terms of cycle time, power and area.

6.3.1 Delay

Computation and checking of the residue is expected to have a delay lower than the ALU itself. Implementing the signatures with residue checking only adds one level of gates in two blocks: (i) the residue generation, and (ii) the checking part. Adding one level of gates in the residue calculation does not impact the critical path delay of the execution stages. Moreover, since the transformation of the residue can be performed in parallel with the checking of the residues, it will not affect the critical path.

Finally, we increase the width of some multiplexors and the bypasses. Although wider multiplexors and bypasses may make them a bit slower, our assessment shows that it is not enough to impact the cycle time.

6.3.2 Area and Power

Area overhead comes mostly from the residue generation and operation hardware. We use previous works for adders and other functional units [13, 18, 22, 37] to estimate the area and power overhead for 2-bit residue for our baseline processor. We assume a 65 nm technology.

Left-hand side of Table 3 summarizes the area overhead. We show in the first column the relative area increase due to the hardware additions required to implement the end-to-end residue. Columns 2 to 4 show the extra area overhead when implementing, on top of residue, the signatures scheme for 2 to 4 bits.

The results show that the overall area increase is small. For 2-bit signatures (with residue), the area increase is 3.2%, 3.7% for 3-bit, and 4.2% for 4-bit signatures.

We have also evaluated the total power (dynamic and leakage) increase due to the proposed framework. We detail the results in the right-hand side of Table 3. For a 2-bit signature protection scheme, we obtained a 3.0% power increase (including the residue protection), 4.0% for 3-bit signatures, and 4.9% when using 4-bit signature. Finally, we show in Table 4 a summary of coverage and cost for every configuration. We find that increasing the size of signatures from 2 to 3 increases coverage considerably at a small cost. However, increasing the signatures' size from 3- to 4-bit gives diminishing returns, since coverage increases marginally, whereas cost increases constantly.

7. RELATED WORK

Mechanisms for off-line testing at pre/post silicon validation rely on some Automated Test Equipment (ATE) [41]. Some works perform tests without any ATE [25], but they focus on off-line testing at fabrication time. Such mechanisms can be used for online testing but they may miss a significant number of faults (since defects are expected to produce faults progressively) or they will manifest only under certain environmental conditions (temperature, voltage, noise, etc.). Smolens et al. [34] use existing scanout chains to measure the wearout. They leverage the observation that wearout faults begin with timing degradation and observe wearout in slightly stressed operation.

Redundancy is a widely used technique as a detection and recovery mechanism for transient and permanent faults in microprocessors. Reis et al. [29, 30] proposed using hardware-

Component	Area				Power			
	Δ%				Δ%			
	Residue	Sign-2	Sign-3	Sign-4	Residue	Sign-2	Sign-3	Sign-4
Byp.	3.1	0.0	1.5	3.0	3.1	0.0	0.0	0.0
FUs	10.0	0.0	0.0	0.0	5.0	0.0	0.0	0.0
UL2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Rename	0.0	25.7	38.6	51.4	0.0	25.7	38.6	51.4
IQ	0.0	1.8	2.7	3.6	0.0	0.8	1.2	1.4
RF	1.2	0.0	1.2	2.4	1.2	0.0	1.2	2.4
DL1	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
ROB	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Alloc.	0.0	1.0	1.0	1.0	0.0	1.0	1.5	2.0
MOB	4.0	0.5	1.5	2.5	4.0	0.5	1.5	2.5
FE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
CLK	N/A	N/A	N/A	N/A	2.5	3.3	3.8	4.4
Total	**2.4**	**3.2**	**3.7**	**4.2**	**1.4**	**3.0**	**4.0**	**4.9**

Table 3: Area and power overheads. Byp. stands for bypasses, Alloc. for allocation logic, FE for the frontend, and CLK for the clock.

Scheme	Coverage	Area	Power
2-bit	75.00%	3.2%	3.0%
3-bit	87.50% ($\Delta = 12.50\%$)	3.7% ($\Delta = 0.5\%$)	4.0% ($\Delta = 1.0\%$)
4-bit	93.75% ($\Delta = 6.25\%$)	4.2% ($\Delta = 0.5\%$)	4.9% ($\Delta = 0.9\%$)

Table 4: Coverage versus overheads

software hybrid schemes which achieve fault tolerance by replicating instructions at compiler level and using hardware fault detectors that make use of this redundancy. The IBM® G5 [36] replicates the frontend and the execution engine; all instructions are executed twice in parallel and compared for error detection. Recovery is achieved keeping a copy of the register file. DIVA [4] uses a simple in-order core as a checker for an out-of-order core.

Re-execution in the same core is used for error detection and recovery from transient faults in several works [8, 11, 21, 24, 26, 31, 35, 40]. The general idea is to use the multithreading capabilities existing in modern SMT processors to run two copies of the same thread. After execution the outcome of the instructions is checked to detect faults and recover from them if possible. This approach incurs some performance degradation. Permanent errors cannot be detected because both threads use the same hardware. The inherent hardware redundancy in CMPs has been also used for error detection and correction. Sundaramoorthy et al. [39] extend Rotenberg's previous work [31] in the context of CMPs. A detailed efficient implementation of CMPs executing redundant threads with recovery capabilities has been described by Gomma et al. [7].

Previous works have already implemented fine-grain online testing. Meixner et al. [16] propose to check invariants during core execution. Signatures have been used in the past to protect the control flow [15]; *a priori* signature is calculated at compile time and inserted in the code. Later, a new signature is generated at run time and compared to the one generated at compile time. This approach implies a non-negligible design cost, due to the required modifications to the ISA, as well as power consumption increase and impact on performance, because of the required signature calculation during runtime. Other proposals are more specific and validate the order in which instructions are executed [27].

Recent works have also focused on light-weight mechanisms to protect the microprocessor pipeline against errors caused by defects and degradation by means of online testing [32]. Notice that compared to our proposal, this approach does not detect either soft or design errors and must rely on other cost-competitive techniques. This work targets a very simple VLIW processor, which lacks most of the complex control logic inherent in today's out-of-order processors. It performs sporadic ad-hoc BIST for different hardware structures (only those exhibiting redundancy in their design). Error checking and computation are not performed concurrently, but rather multiplexed in time. This means that testing will be performed at idle cycles, and an error will be detected after it has manifested and affected the microarchitecural state. Therefore, it requires complementary mechanisms to roll-back to previous computational epochs once an error is detected. However, an epoch can not be started before the full testing has been performed, which implies performance overhead.

Finally, a recent work [17] verifies the data flow by means of compiler support, similar to previous works used for control flow concurrent testing [15]. This novel approach allows validating the data flow for those blocks of code that have a statically known data-flow graph (basic blocks). For each basic block, the data flow for each logical register across all the instructions within the basic block is computed. This information is check-summed and embedded into the program binary. During execution, the hardware computes the "a-posteriori" hash signature and in case of a mismatch, an error can be detected. One of the problems with this approach is that the compiler assumes an initial data flow state at the beginning of every basic block, and the processor must guarantee that the data flow for each register is reset to the initial value. However, this is complex to implement on out-of-order processors and can induce coverage loss under some scenarios. The cost of such approach is similar to those that protect the control flow based on the compiler help: a new ISA is required, the program binary is increased which causes a performance slowdown due to

increased number of fetches, and only the user code (which needs to be recompiled) is protected.

8. CONCLUSIONS AND FUTURE WORK

New process generations constrain burn-in capabilities, thus increasing the probability of having in-the-field error rates. Moreover, sources of transient errors such as erratic bits and soft errors remain a challenge. Parity and ECC are effective to detect soft and hard errors in SRAM bit-cells, but are unable to protect logic structures.

This paper proposes a low-cost *online end-to-end* protection scheme that guarantees that data consumed through register dependencies is correct. The proposed end-to-end scheme allows detecting hard errors, errors caused by degradation, intermittent errors such as soft errors and design errors (bugs). Our design leverages the microarchitectural knowledge of the runtime behavior, and it is independent of the particular processor implementation. The technique relies on a small set of signatures that sign each produced value. Furthermore, we describe how to set up an end-to-end residue coding scheme, and propose a way to integrate it with little additional cost.

Overall, our design is able to protect the *rename* tables, *wake-up* logic, *select* logic, *input multiplexors*, *operand read and writeback*, the *register free list*, and the *replay* logic. By combining it with an end-to-end residue coding scheme, we extend the protection to the *functional units*, *MOB* data and addresses, and the *register file* storage. Our results show a coverage around 87% for 3-bit signatures and 2-bit residue, with a small cost in power and area of 4% for the whole core.

As future work, we plan to extend our framework to cover also the rename logic by generating signatures in the rename stage but in a procedure completely decoupled to the rename process, in such a way that if the rename logic fails to enforce a correct data flow it will be detected by a signature mismatch. We also plan to work on extending the current design to improve the validation phase.

Acknowledgments

We would like to thank Sorin Iacobovici and Abhijit Jas from Intel for the interesting and thorough discussions while we elaborated this work.

This work has been partially supported by the Spanish Ministry of Education and Innovation under grant TIN2007-61763.

9. REFERENCES

[1] J. Abella, P. Chaparro, X. Vera, and J. Carretero. On-line failure detection and confinement in caches. In *Proceedings of the 13th IEEE International On-Line Testing Symposium (IOLTS)*, 2008.

[2] M. Agostinelli, J. Hicks, J. Xu, B. Woolery, K. Mistry, K. Zhang, S. Jacobs, J. Jopling, W. Yang, B. Lee, T. Raz, M. Mehalel, P. Kolar, Y. Wang, J. Sandford, D. Pivin, C. Peterson, M. DiBattista, S. Pae, M. Jones, S. Johnson, and G. Subramanian. Erratic fluctuations of SRAM cache vmin at the 90nm process technology node. In *Technical digest of IEEE International Electron Devices Meeting (IEDM)*, pages 655–658, December 2005.

[3] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3ghz fifth generation SPARC64 microprocessor. In *DAC '03: Proceedings of the 40th Conference on Design Automation*, pages 702–705, 2003.

[4] T. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of International Symposium on Microarchitecture (MICRO)*, 1999.

[5] A. Avizienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. *IEEE Transactions Computers*, C–20(11):1322–1331, 1971.

[6] T. Barnett, A. Singh, and V. Nelson. Extending integrated-circuit yield-models to estimate early-life reliability. *IEEE Transactions on Reliability*, 52(3):296–300, Sept. 2003.

[7] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, 2003.

[8] M. Gomaa and T. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the 32nd annual International Symposium on Computer Architecture (ISCA)*, 2005.

[9] G. Hinton, D. Sager, M. Upton, D. Bogs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium® 4 processor. *Intel Technology Journal*, 5(1):13, Feb. 2001.

[10] Y. Hsing, C. Wang, C. Wu, C. Huang, and C. Wu. Failure factor based yield enhancement for SRAM designs. In *Proceedings of the 19th International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, pages 20–28, Oct. 2004.

[11] S. Kumar and A. Aggarwal. Reducing resource redundancy for concurrent error detection techniques in high performance microprocessors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[12] S. Kundu, T. Mak, and R. Galivanche. Trends in manufacturing test methods and their implications. In *Proceedings of the International Test Conference (ITC)*, pages 679–687, 2004.

[13] G. Langdon and C. Tang. Concurrent error detection for group look-ahead binary adders. *IBM Journal of Research and Development*, 14(5):563–573, 1970.

[14] J. Lo. Reliable floating-point arithmetic algorithms for error-coded operands. *IEEE Transactions on Computers*, 43(4):400–412, 1994.

[15] A. Mahmood and E. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb. 1988.

[16] A. Meixner, M. E. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the 40th International Symposium on Microarchitecture (MICRO)*, 2007.

[17] A. Meixner and D. Sorin. Error detection using dynamic dataflow verification. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 104–118, 2007.

[18] S. Mitra and E. McClusky. Which concurrent error detection scheme to choose? In *Proceedings of the International Test Conference (ITC)*, 2002.

[19] P. Monteiro and T. Rao. A residue checker for arithmetic and logical operations. In *2nd Fault Tolerant Computing Symposium*, 1972.

[20] M. Mueller, L. C. Alves, W. Fischer, M. L. Fair, and I. Modi. RAS strategy for IBM S/390 G5 and G6. *IBM Journal of Research and Development*, 43(5):875–888, 1999.

[21] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th annual International Symposium on Computer Architecture (ISCA)*, 2002.

[22] A. Pan, J. Tschanz, and S. Kundu. A low cost scheme for reducing silent data corruption in large arithmetic circuits. In *Proceedings of International Symposium on Defect and Fault Tolerance of VLSI Systems (DFTVS)*, 2008.

[23] N. Quach. High availability and reliability in the Itanium processor. *IEEE Micro*, 20(5):61–69, Sept.-Oct. 2000.

[24] M. Qureshi, O. Mutlu, and Y. Patt. Microarchitectural-based inspection: a technique for transient-fault tolerance in microprocessors. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, 2005.

[25] R. Rajsuman. RAMbist builder: a methodology for automatic built-in self-test design of embedded RAMs. In *Proceedings of the International Workshop on Memory Technology, Design and Testing (MTDT)*, page 50, 1996.

[26] J. Ray, J. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, pages 214–224, 2001.

[27] V. Reddy, A. Al-Zawawi, and E. Rotenberg. Assertion-based microarchitecture design for improved fault tolerance. In *Proceedings of International Conference on Computer Design (ICCD)*, pages 362–369, 2007.

[28] K. Reick, P. Sanda, S. Swaney, J. Kellington, M. Floyd, and D. Henderson. Fault-tolerant design of the IBM Power6™ microprocessor. In *Proceedings of the Hot Chips 19 Symposium*, 2007.

[29] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2005.

[30] G. Reis, J. Chang, N. Vachharajani, R. Rangan, D. August, and S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005.

[31] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the Annual International Symposium on Fault-Tolerant Computing (FTC)*, page 84, 1999.

[32] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra low-cost defect protection for microprocessor pipelines. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 73–82, 2006.

[33] Sih and Reinheimer. Checking logical operations by residues. *IBM Technical Disclosure Bulletin*, 15(7):2325–2327, 1972.

[34] J. Smolens, B. Gold, J. Hoe, B. Falsafi, and K. Mai. Detecting emerging wearout faults. In *Proceedings of the 3rd Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2007.

[35] J. Smolens, J. Kim, J. Hoe, and B. Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, 2004.

[36] L. Spainhower and T. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective. *IBM Journal of Research and Development*, 43(5/6):863–873, 1999.

[37] U. Sparmann and S. Reddy. On the effectiveness of residue code checking for parallel two's complement multipliers. *IEEE Transactions on Very Large Scale Integration Systems*, 4(2):227–239, 1996.

[38] SPECCPU 2000. SPEC Newsletter, Sept. 2000.

[39] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the 33th International Symposium on Microarchitecture (MICRO)*, 2000.

[40] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002.

[41] D. Wu, M. Lin, M. Reddy, T. Jaber, A. Sabbavarapu, and L. Thatcher. An optimized DFT and test pattern generation strategy for an intel high performance microprocessor. In *Proceedings of the International Test Conference (ITC)*, pages 38–47, 2004.