Tolerance of Performance Degrading Faults for Effective Yield Improvement

Tong-Yu Hsieh[1], Melvin A. Breuer[2], Murali Annavaram[2], Sandeep K. Gupta[2] and Kuen-Jong Lee[1]
[1]*Dept. of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan*
[2]*Dept. of Electrical Engineering, University of Southern California, Los Angeles, USA*

Abstract

To provide a new avenue for improving yield for nano-scale fabrication processes, we introduce a new notion: ***performance degrading faults (pdef)****. A fault is said to be a pdef if it cannot cause a functional error at system outputs but may result in system performance degradation. In a processor, a fault is a pdef if it causes no error in the execution of user programs but may reduce performance, e.g., decrease the number of instructions executed per cycle. By identifying faulty chips that contain pdef's that degrade performance within some limits and binning these chips based on the their resulting instruction throughput, effective yield can be improved in a radically new manner that is completely different from the current practice of performance binning on clock frequency.*

To illustrate the potential benefits of this notion, we analyze the faults in the branch prediction unit of a processor. Experimental results show that every stuck-at fault in this unit is a pdef. Furthermore, 97% of these faults induce almost no performance degradation.

1. Introduction

Permanent faults have classically been put into several categories, some of which are listed below, based on how their effects are manifested.

1. Redundant – faults which, when present, can not be detected via a full scan DFT methodology.
2. Error-producing but non-functionally activated – faults that produce erroneous responses only when certain test data is applied, but this data cannot occur during the normal operation of the system.
3. Error-producing and functionally activated – faults that produce erroneous results for some normal operations.
4. Error-tolerant – these are error-producing faults where the output responses from the system are acceptable to the end user, even if some are erroneous [1-4].

When one of the above types of faults occurs in a newly manufactured die, one or more of several actions are taken. Usually faults in category 1 above are ignored, and in most cases, one does not even know of their existence.

Normally faults in categories 2 and 3 that are detected by the die/chip-test process are not differentiated. Typically, such faults are handled in one or more of the following ways.

- The die is discarded.
- The effect of the fault is masked by using some form of redundancy, such as extra rows of memory in a cache. Here, the row containing the fault is logically replaced by a spare row. In this case, there is almost no impact on the normal operation of the die. Error Correcting Code (ECC) is also used to mask the effect of errors.
- The effect of the fault is eliminated by logically removing a part of the system's logic that contains the fault [5-6]. For example, if the fault lies in the first quarter of a cache of size 4K blocks, the system can be reconfigured to have only 3K blocks available to the end user. In such a case, the chip must be given a new part number and sold at a discounted price.
- The effect of the fault is eliminated by operating the chip at a lower clock-rate or higher voltage.

Before we extend the above classification by introducing our new notion of performance degrading faults, consider a computing system characterized by its functionality and performance. The functionality is defined in terms of logic- or higher-level requirements on the input-output behavior of the system. For some systems, functionality may be specified in terms of their logic-level input-output behavior described using, say, state transition graphs. Most modern processors execute a program in a non-deterministic manner but guarantee the functionality in terms of the output they produce for any given input. In such systems, it is important to distinguish between the circuit-level and architecture-level abstractions. At the circuit level, the functionality of each sub-module is deterministic and its performance may be described in terms of parameters such as the worst-case delay of logic blocks, measured in pico-seconds. In contrast, at the architecture level the operation may be non-deterministic for a given program while guaranteeing certain input-output behavior. As a performance metric, at the circuit level we might consider the worst case timing path for each block, while at the architecture level we might consider the number of instructions retired per cycle.

The primary focus of this paper is to exploit the gap between functional correctness and performance correctness by treating errors in logic either as functional errors or performance errors. We assume that the logic portion of a die is designed using the popular full-scan design-for-test (DFT) methodology. Here, during the test mode, each flip-flop is configured to be in one of several shift registers (i.e., *scan registers*), and arbitrary test data loaded into these devices. Using a normal functional clock, the circuit's response to this test data can be captured into these devices, and then shifted-out for observation. By using a rich enough set of test data, and special clocking

978-1-4244-4867-8/09/$25.00 ©2009 IEEE

techniques, most non-redundant faults can be detected, and to some extent, their location isolated to within a few gates.

In this form of testing, if an error occurs in some flip-flop, then such a fault will be in category 2 or 3, even though the effect of the fault might not create an error at the output of the device. It thus seems like there may be an interesting category of faults that have not been adequately studied. We refer to this new class of faults as performance degrading faults. Before we precisely define this new category of faults, consider an illustrative example.

Example 1: Consider the portion of a large circuit shown in **Figure 1**. This part of the circuit is intended to carry out the following function. The 32-bit register R_2 transmits data over a long bus to R_3 using some special form of signaling, where less power is dissipated if the data sent at time t is similar (small Hamming distance) to that at the previous time, t - 1. Thus, if R_2 holds the current datum being transmitted, and R_1 holds the next piece of datum to be transmitted, then the Hamming distance between these two pieces of datum can be computed in the block DISTANCE, and if greater than 16, a logic 1 is applied to the control input of the MUX; otherwise a 0 is applied. The MUX then selects the true or complemented value of data in R_1 to send to R_2. One extra flip-flop in R_2 and one in R_3 keep track of whether the data is transmitted in true or complement form.

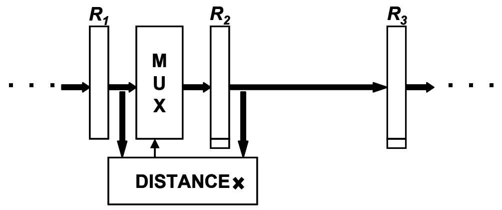

Figure 1: A performance degrading fault

Now assume that a fault occurs in the block DISTANCE so that its output is sometimes in error. The result is that sometimes the true rather than the complemented value of datum is transmitted or vice-versa, resulting in more power consumption, thus a loss in some form of performance. However, the datum R_3 receives is not in error. In fact, the output of DISTANCE can be stuck-at 0 or 1 and R_3 will still receive the correct data.

We semi-formally define a ***performance degrading fault*** (pdef) with respect to a circuit S as a fault that: (1) can produce significant errors, such as permanent faults in storage cells at the circuit level, or a deviation in program execution for a processor (compared to a fault-free version of the processor under identical conditions); (2) does not produce any errors in the normal functional outputs of S, such as the final output produced for a user's program; (3) is not masked or compensated for by reconfiguration or modifying how S is operated, e.g., by replacing the faulty module by an available spare or by increasing the supply voltage; but (4) does adversely impact some aspect of system performance, such as throughput, latency or power. A pdef is said to pass a test if the degradation it introduces is below a threshold level; otherwise it is said to fail the test. The threshold is determined by engineers and marketing personnel familiar with factors such as the product being produced and the applications in which it will be used.

Referring to Example 1, the fault in the block DISTANCE is a pdef that might be acceptable if the output of DISTANCE is wrong less than 5% of the time.

In this paper, we introduce and develop the notion of performance degrading faults. We demonstrate that a large fraction of modules in modern processors are such that the notion of pdef's is applicable. Furthermore, we carry out a detailed analysis of faults in one such module, namely a branch predictor, and show that all faults in this module are pdef's and most of them cause acceptable degradation in system performance. This paper demonstrates that further systematic study of pdefs can provide significant yield benefits for the foreseeable future.

In [7] some effects of performance faults in a branch predictor are described. However the main idea of [7] is to employ redundant hardware to reduce the performance degradation, which is different from our work that employs no redundancy. Performance degrading effects caused by soft-errors are discussed in [8], which is also different from our work that deals mainly with hard-errors.

2. Aspects of High-Performance Processors

As the number of transistors on a die continue to increase following Moore's law, it has become imperative to find meaningful ways to use these transistors. Early techniques such as pipelining provided significant improvements to processor performance. But as the performance gains from these techniques saturated, superscalar Out-of-Order execution was introduced as the next leap in microarchitecture advancements. Most of the current high performance processors, such as the Intel Core i7, AMD Opteron and IBM Power 7, fall under this category. These processors fetch, issue, execute and retire multiple instructions every clock cycle. Another interesting aspect to note is that these processors may execute instructions out of program order but will always guarantee functional correctness by making the result of execution seen by an external observer to be in program order. To exploit such instruction level parallelism, several hardware structures were introduced in chip design.

- Caches were introduced to address the growing gap between processor clock and memory access latency. Currently about 50% of the die area is dedicated to L1 and L2 caches [9].
- Branch predictors are used to predict the outcome of a branch instruction before the branch instruction is executed. Deep pipelining and super scalar execution made it critical to accurately predict branch instructions so as to speculatively fetch the instructions that need to be executed following a branch. These predictors only predict the direction of a branch. If a branch is predicted to be taken, a branch

target buffer is then used to predict the target address of the branch. Both the predictor and the target buffer are only used in speculative execution.
- Speculative instruction scheduling is a technique to eagerly issue instructions to execution units even before having all the input operands ready. In this technique the scheduling hardware predicts when the input operand data is likely to be available.
- More recently, due to increasing power consumption concerns, processor designers created multiple power states, typically called C-states, that gradually reduce the power consumption by reducing the chip functionality. For instance, Intel Core i7 supports at least six C-states to allow the software fine grain control of processor power consumption [9]. The circuitry for managing the C-states is quite complex and the complexity is expected to increase as more C-states are introduced.

The above list is only a representative sample of hardware blocks that are used in current processors. One interesting aspect that is common amongst all these blocks is that these blocks are used only to improve performance, often via speculative instruction execution. The speculation is verified later in the pipeline and on a mis-speculation the mis-speculated instructions are simply discarded without any impact on the functional correctness of the program. However, mis-speculation degrades architecture-level performance and increases execution time. We will refer to these speculative execution support blocks as performance enhancing non-code processing (PENCoP) circuitry (pronounced "pen cop"). It has been observed that PENCoP circuitry occupies about 50% -70% of the die area of a high-performance processor [9].

The proposed notion of performance degrading faults is applicable to PENCoP circuitry. In particular, many faults in a PENCoP block may be pdef and a large fraction may only cause acceptable performance degradation. (Since a PENCoP block is purely optional from the point of view of functional correctness, all faults within might seem to be pdef. This is **not** always the case. In the case study reported in this paper we do take this fact into account.) This opens up the possibility of improving yield, by enabling a completely new type of binning based on architecture-level performance. In particular, this type of performance binning is completely different from the clock-frequency binning that is used by microprocessor manufacturers.

Using, for example, the 50% figure for PENCoP circuitry, assume that 30% of the single stuck-at faults in such circuitry can be classified as performance degrading faults. Also, assume that only stuck-at faults exist, and 40% of manufactured die have a single stuck-at fault, and 12% have two or more faults. Ignoring redundant faults and using a scan test methodology, (40+12=) 52% of the die would be classified as having faults and be discarded. Of the 40% of die that have a single failure, the failures in 50% of these die, on average, are in the PENCoP circuitry, and 30% of these are pdef's. So, if these die are packaged, relabeled and sold, the effective yield increases from 0.48 to $0.48 + 0.50 \times 0.40 \times 0.30 = 0.54$, resulting in a 12.5% increase in yield. Of course, this is a phenomenal result based on many assumptions, but it certainly implies that further analysis is warranted.

In the remainder of this paper, we will carry out a detailed case study of one PENCoP module that is used in most modern processors, namely the branch-predictor. The goal of this case study is to quantify the key parameters that can be used to estimate the yield improvement enabled by exploitation of the proposed notion of pdef.

3. Case Study

In this section, we present a detailed analysis of branch prediction circuitry. Based on the analysis results, we also identify which faults in the branch predictor lead to acceptable architecture-level performance.

In computer architecture, a branch predictor is a micro-architectural block that determines whether a conditional branch in the instruction flow of a program is likely to be taken or not. In deeply pipelined superscalar processors multiple instructions are fetched every clock cycle. For instance, in the Core i7 processor, four instructions are fetched every clock from the instruction cache into the processor core [9]. On average one in every five fetched instructions is a branch instruction. Hence, one expects to encounter one branch instruction every other fetch cycle. However, the branch instruction B_1 fetched in cycle C_1 is only decoded in cycle $C_1 + 1$ (assuming that decode occurs one cycle after fetch). Since the branch direction is only known at the time of execution of the instruction, the decode stage of the pipeline at time $C_1 + 1$ does not know if the branch instruction fetched in cycle C_1 is a *taken* or *not-taken* branch. To address this problem the branch instruction in cycle C_1 accesses a branch predictor to determine if that branch is likely to be taken or not taken. If it is a *taken* branch, a branch target buffer is then accessed to predict the exact target address. Thus there are two levels of prediction that are made: first to predict the branch direction, and then to predict the target address. In the case of a predicted taken branch, the fetch logic in the pipeline brings instructions from the predicted target address. However, B_1 is executed n cycles after it is fetched (assuming the pipeline depth is n cycles between fetch and execute). At this time the processor determines the *actual* branch direction and target address of B_1. The branch prediction verification logic then compares the actual direction with the predicted direction. If there is a match, then it compares the actual with the predicted target address. If both match then nothing needs to be done in the pipeline. However, if the prediction is incorrect, the processor flushes all the instructions that were fetched after B_1 and resteers the fetch stage to fetch instructions from the actual target address.

There is one important aspect to the operation of the branch predictor that is of interest to our research, namely error resilience is inherent. On a misprediction, the processor pipeline must be flushed and instructions from the correct path are fetched for execution. Therefore, mispredictions lead to performance loss but not erroneous end-results. Good prediction schemes exist that predict the correct branch direction over 93% of the time [10].

3.1 High-Level Description

Conceptually, one might think that there exist faults that could occur in a branch prediction unit that cause its misprediction rate to increase. For a chip with such a fault, the pipeline would have to be flushed more often and the throughput of the process would consequently decrease. However, no erroneous results would be produced. To quantify this concept we simulated several large benchmark programs. The processor employed a *gshare* branch prediction unit [10], shown in **Figure 2**, that achieves an average accuracy of **94%** on a set of benchmarks shown in **Table 1**. These benchmarks are all taken from the CPU2000 benchmark suite commonly used in the processor industry to benchmark relative performance [11]. The total instructions executed in each benchmark are listed in **Table 1** (see Column #instructions), along with the total number of branches among these instructions (see Column #branches). Note that since unconditional branches, i.e., *jump* instructions, must always be taken, the branch predictor only predicts directions for conditional branches which are taken only when a certain condition is satisfied. The direction prediction accuracy thus is evaluated for only conditional branches, and the numbers of these branches are also shown in **Table 1** (see Column #conditional).

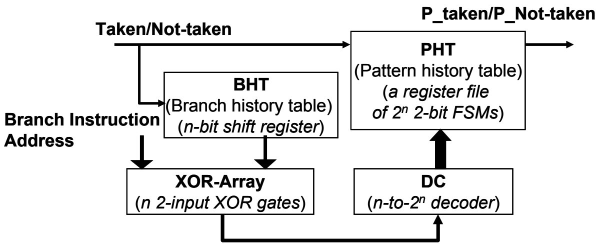

Figure 2: Main structural blocks in a *gshare* branch prediction unit

The BHT refers to the first-level branch history table, and consists of an n-bit shift register. It records the direction taken, encoded as a 0 or 1, by the last n branch instructions executed. These n instructions might all be the same branch instruction, or different ones. The contents of this register are used to hash the low-order n bits of the address of the current branch instruction to the address of a finite state machine in a second-level pattern history table, PHT. (Bits 0 and 1 are not considered since they are both always 0 in a 32-bit processor architecture and thus are unlikely to generate a good hashing result. Bits 2 to 19 are thus used for hashing.) One simple hashing function is to XOR the j'th bit of the address of the branch instruction with the j'th bit of the BHT register. The 1-hot decoder (DC) is used to select one of the 2^n 2-bit finite state machines that make up the PHT. The state diagram for each of these FSMs is shown in **Figure 3,** which is basically a 2-bit saturation counter. For the circuit being considered, n=18.

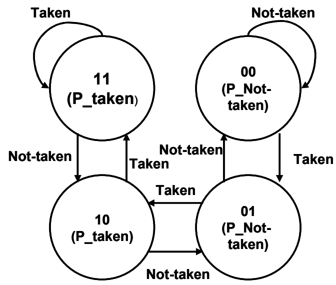

Figure 3: State diagram for a 2-bit saturation counter in the PHT

Assume that a program is in a long loop where a branch instruction is consistently *taken* (Taken=1). In this case, one would find the machine in the state encoded by 11. Once in this state, if a branch is *not-taken*, but followed by many *taken* branches, then the FSM goes into the state 10 and then returns to the state 11. If three or more consecutive *not-taken* branches occur, then the FSM will be in state 00.

When the processor encounters a branch instruction, the fetch stage of the pipeline must determine the direction using this *gshare* predictor. By hashing the branch instruction address with history of the last n branches, the predictor selects one FSM entry using the decoding logic. The most significant bit of the two-bit FSM provides the prediction. If the MSB is 0 it is predicted *not taken*, and if the MSB is 1 it is predicted *taken*. In addition to using the predictor during the fetch stage, the predictor is also accessed after the execution of each branch instruction to update the BHT and the two-bit FSM. If the branch is actually taken then a value of 1 is shifted into BHT, otherwise 0 is shifted into BHT to keep track of the last n branch outcomes. The FSM that provided the prediction is incremented by 1 for a *taken* branch unless its state is already 11; it is decremented by 1 for a *not-taken* branch unless its state is already 00.

Table 1: Five CPU2000 benchmarks and the prediction accuracies using gshare

Benchmark	Usage	# instructions	# branches	# conditional	Accuracy
twolf	Simulate Place & route	258,754,002	28,802,191	21,613,566	0.9202
mcf	Combinational optimization	259,641,194	43,844,903	31,245,377	0.9304
vpr **(*route*)**	FPGA routing	692,798,513	76,708,172	63,948,985	0.9381
gap	Interpreter	1,169,577,655	167,343,246	119,351,547	0.9729
gzip	Compression	3,367,270,629	351,442,008	256,353,095	0.9394

3.2 Logic Level Description of Branch Predictor

To inject faults, we implemented the branch prediction unit using gates and flip-flops, and then described this design in the C-programming language. Signal $A = 1$ if the **actual** branch is taken, otherwise $A = 0$. Similarly, $P = 1$ if the branch is **predicted** *taken*, otherwise $P = 0$. The BHT is an 18-bit left-shift register, and the XOR array consists of 18 2-input XOR gates; the inputs to the i'th gate being the output of the i'th cell in the BHT and the i'th bit of the address of the branch instruction. The output of this hashing circuitry is used to select one of the 2^{18} FSMs, identified by setting the signal *select(j)* = 1.

To minimize its complexity, we implement the 18-input 2^{18}-output decoder as a two-stage pre-decoded decoder. The first stage has two 9-input 2^9-output decoders. Each of these two decoders is driven by half of the original 18 address bits and, for any input address, each selects appropriate one of its 2^9 outputs by making it low. The 2^9 outputs of each of the first-stage decoders are inputs to the second stage, which is an array of 2^{18} two-input NOR gates. Each NOR gate's output is an output of the decoder. **Figure 4** shows one FSM cell in the pattern history table. If a prediction is to be made, then $Z(j)$-bar indicates the result, while $Z(k) = 1$ for all $k \neq j$.

The FSM is accessed in two modes, namely, to make a prediction or to update once the actual disposition of a previous prediction becomes known. In both cases, the decoder is used to access the FSM. In the first case, the flip-flops in the FSM are not clocked; only the current content is used to make the prediction. In contrast, in the second case the flip-flops are clocked to update the FSM.

In terms of the numbers of gates and wires (and hence in terms of layout area), *gshare* design is dominated by the decoder (DC) and the pattern history table (PHT).

3.3 Faults and Test Cases

We study single stuck-at 0 and 1 faults. Due to the large size of the circuit, the set of all single stuck-at faults contains over 2×10^7 faults. Application of classical notions of equivalence and dominance leaves a number of faults that is still impractically large to simulate for benchmarks containing millions of instructions.

Recall that two distinct faults associated with a block of combinational logic are said to be **equivalent** if each associated faulty circuit produces the same output when stimulated by the same input, and for all possible inputs. For example, a stuck-at 0 fault at the input A to a 2-input XOR gate is not equivalent to a stuck-at 0 fault at B. However, these faults might be considered as being **isomorphic or permutation equivalent**, i.e., by relabeling the inputs we can use tests for faults at A as tests for faults at B.

The branch prediction circuitry has a great deal of symmetry in its structure and many of its outputs drive circuits that are functionally isomorphic. In particular, the 2^{18} FSMs in the PHT are functionally isomorphic as (a) each implements the same state transition graph (STG), and (b) a linear function (implemented by an array of n XOR gates) is used to hash each branch instruction's address by a history of the most-recent n branches stored in the BHT. Hence, a fault in one FSM is likely to have a similar overall impact on architecture-level performance as the same fault in any other FSM. Hence, the notion of isomorphic or permutation equivalence is applicable to the largest parts of this circuit, namely the decoder and the FSMs. Therefore, even though this circuit contains over 2×10^7 single stuck-at faults, very few of these need to be explicitly analyzed/simulated to determine the effects of these faults on performance. (Nonetheless, we target similar faults in multiple FSMs. The results will show that such detailed analysis is not needed as each instance of a similar fault results in a similar reduction in performance.)

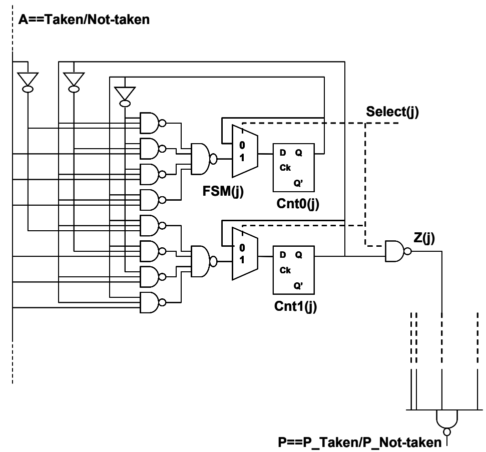

Figure 4: Logic implementation of each FSM in the PHT of the *gshare* predictor

We focus on just 32 lines, and thus we have 32 stuck-at 0 and 32 stuck-at 1 faults. **Table 2** shows the location of the injected faults, as well as each of their "line index". Thus line-index 1 refers to the 2nd bit of the branch instruction address, where both a s-a-0 and s-a-1 will be injected.

3.4 Experimental Results

Using the five benchmark programs shown in **Table 1**, we have carried out two sets of experiments to evaluate the impact of these faults on system performance. In both sets of experiments, we execute the benchmarks on a fault-free and each of the faulty versions of the processor. In the first set, we measure the ***accuracy of branch-prediction***, i.e., the percentage of branch instructions for which *taken*/*not taken* is accurately predicted. Note that the accuracy of prediction is evaluated only considering the branch instructions that are completely executed, i.e., not flushed due to misprediction. Since these branches are executed in the fault-free as well as every faulty case, the degradation in prediction accuracy can be fairly compared. In the second set of experiments, we compute the ***number of cycles required to complete execution of benchmarks***. Since the complexity of such simulations is higher, we conduct this analysis for a smaller number of cases.

Table 2: Location of injected faults

Line index	Content
1	2nd bit of the branch instruction address
2	11th bit of the branch instruction address
3	19th bit of the branch instruction address
4	0th bit of the BHT
5	9th bit of the BHT
6	17th bit of the BHT
7	0th input bit (from BHT) to the XOR array
8	9th input bit (from BHT) to the XOR array
9	17th input bit (from BHT) to the XOR array
10	0th bit of the output of the XOR array
11	9th bit of the output of the XOR array
12	17th bit of the output of the XOR array
13	0th bit of output of the decoder
14	100th bit of output of the decoder
15	1000th bit of output of the decoder
16	10000th bit of output of the decoder
17	100000th bit of output of the decoder
18	262143rd bit of output of the decoder
19	MSB of the 0th FSM
20	MSB of the 100th FSM
21	MSB of the 1000th FSM
22	MSB of the 10000th FSM
23	MSB of the 100000th FSM
24	MSB of the 262143rd FSM
25	LSB of the 0th FSM
26	LSB of the 100th FSM
27	LSB of the 1000th FSM
28	LSB of the 10000th FSM
29	LSB of the 100000th FSM
30	LSB of the 262143rd FSM
31	P (predicted direction taken)
32	A (actual direction taken)

3.4.1 Branch Prediction Degradation

We first address the issue of how much branch prediction accuracy is lost due to a fault. We employed the C-programming-language-based *SimpleScalar* simulation tool [11] that mimics the execution of a program. The above implementation of *gshare* unit is integrated in *SimpleScalar*. When a branch instruction is to be executed, the address of the branch instruction is recorded and processed by the C-model of the branch prediction circuitry, including an injected fault (if any). When the simulator determines the correct direction of a branch, namely the value of *A*, this data is also recorded and processed by the C-model of the branch predictor.

We will discuss the impact of these faults, one at a time, with respect to a single benchmark program, namely *twolf*. Later we will show the results for other benchmark programs. The *x*-axis of **Figure 5** shows the line-index of the fault, as given in **Table 2**, and the *y*-axis shows the percent degradation in prediction accuracy. For the fault-free case, the predictor is correct 92.02% of the time. Note that the *y*-axis of **Figure 5** is scaled such that the maximum loss of accuracy depicted in 1%. We have done so, since few faults cause significant degradations (see caption of the figure for exceptions).

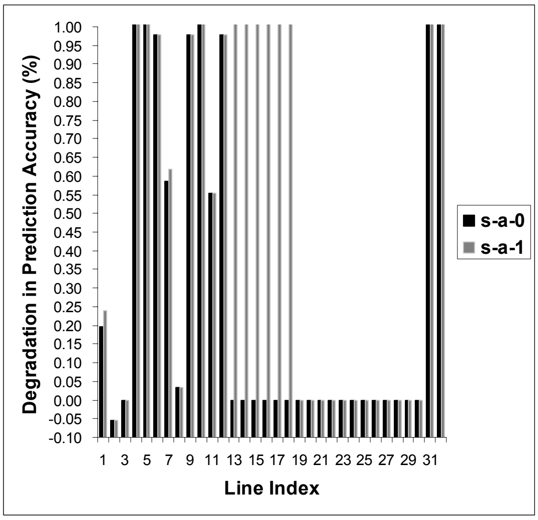

Figure 5: Degradation in branch prediction accuracy for single stuck-at faults and *twolf* benchmark. (Degradations for stuck-at faults at Line indices 4, 5, 10, 31 and 32 and stuck-at-1 faults at line indices 13 to 18 are larger than 1% and not shown accurately.)

The number of gates and lines in the decoder and the PHT collectively constitute 99+% of the total numbers of gates and lines in the entire *gshare* design. Hence, we start by describing the results for faults associated with these two modules. Also, for these two modules, we follow our analysis of the faults shown in **Table 2**, by carrying out a more detailed analysis of the faults within the modules.

A. Decoder

We considered faults associated with the output of the decoder. Since the 2^{18} decoder outputs are 1-hot coded, we expected that a s-a-0 on an output will rarely cause an error, and when it does it results in a branch being predicted *not-taken*. A s-a-1 fault usually results in two FSMs being selected, and results in a branch being predicted *taken* if either of these FSMs predict a branch as *taken*. Also, assuming most of the FSMs are used, and that the hash addresses are uniformly distributed over the 18-bit address space, each select line would be equally likely to be activated. Finally, we expected faults associated with line index 13 to be permutation equivalent to the corresponding faults with line index values of 14 to 18.

Our experimental results confirm our understanding of how faults affect the functionality of this circuit. For the six lines selected to have a fault as specified in **Table 2**, each single s-a-0 fault led to no degradation in prediction accuracy, while each s-a-1 fault resulted in 18.27% reduction in accuracy.

We followed the above high-level analysis with simulations of faults internal to the decoder. As mentioned earlier, our implementation is an area-efficient two-level pre-decoded decoder. We injected stuck-at faults at the inputs and outputs of level-1 and level-2 decoders and at the inputs and outputs of each gate. Half of these faults cause none of the outputs to be selected while the other half cause two outputs to be selected, one of which is the output selected in the fault-free version. In particular, while a s-a-1 fault at an output causes an additional *fixed* decoder output to be selected (in addition to the output selected in the fault-free version) with a probability close to 1, some internal faults can cause a *different* output to be selected (in addition to that selected in the fault-free version) with probabilities approximately equal to 1, 1/2, or $1/2^9$.

Table 3: Location of faults internal to the decoder

Line type	Location	Fault effect s-a-0	Fault effect s-a-1
A	An address input of a level-1 decoder	The selected output is inactivated with probability ≈ **0**	An additional **different** output is activated with probability ≈ **0.5**
B	An input of a gate in a level-1 decoder	The selected output is inactivated with probability ≈ **0**	An additional **different** output is activated with probability ≈ $1/2^9$
C	An address input of the level-2 decoder	An additional **different** output is activated with probability ≈ **1**	The selected output is inactivated with probability ≈ **0**
D	An input of a NOR gate in the level-2 decoder	An additional **fixed** output activated with probability ≈ $1/2^9$	The selected output is inactivated with probability ≈ **0**
E	An output of the level-2 decoder	The selected output is inactivated with probability ≈ **0**	An additional **fixed** output is activated with probability ≈ **1**

We analyze the effects of faults within the decoder by classifying the lines of the decoder into five categories according to fault locations, as shown in **Table 3**. We then randomly select six lines belonging to each of these five categories and inject faults, respectively, indexed by A_1, A_2, …, A_6, B_1, …, E_6. Note that the lines selected for category E are exactly the same as those shown in **Table 2** with line index values from 13 to 18.

Figure 6 shows the degradation in prediction accuracy caused by faults within the decoder. The faults with probabilities close to 1 of selecting two decoder outputs, i.e., at line categories C and E, can significantly degrade the accuracy, up to 18.3%. The faults that select two decoder outputs with much smaller probabilities, i.e., at line categories B and D, as well as faults that cause no decoder output to be selected, have no or negligible impact (0.04% reduction) on the prediction accuracy. Fortunately, most of the faults in the decoder belong to these categories. Therefore ***approximately 83% of faults in the decoder cause almost no degradation in prediction accuracy***.

B. FSMs

Similar to the case for a decoder line being s-a-0, the impact of either flip-flop of an FSM being faulty is essentially zero. This is confirmed by our results shown in **Figure 5** for all faults associated with the FSMs, namely line index values 19 to 30.

This implies that ***any fault within the FSM also has a negligible impact on the prediction accuracy***. This follows because even if one of the saturation counters operates incorrectly, its output is rarely sampled and hence this erroneous operation has little impact on the system.

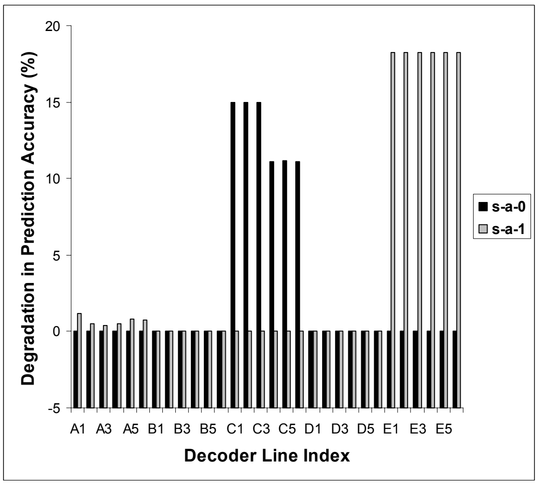

Figure 6: Degradation in branch prediction accuracy for single stuck-at faults at lines within the decoder, for *twolf* benchmark.

C. Other Modules

The Instruction Address: A stuck-at fault (SAF) on line-index 1 results in an error in the 2nd bit of the instruction address approximately 50% of the time. Such an error propagates through the hash function and results in selecting the wrong FSM. The s-a-1 (0) fault leads to an insignificant degradation of about 0.24% (0.20%). A SAF on line-index 2, namely the 11th bit in the instruction address, actually leads to a slight increase in branch prediction accuracy, an indication of the fact that *gshare* branch-predictor is not optimal and hence some faulty version may provide higher accuracy for some benchmark. (The increase in accuracy is small and occurs only for a few cases.) Finally, a SAF on line-index 3 results in almost no degradation. This is because most addresses have a 0 in the MSB; hence a s-a-0 fault on this line produces no fault effect and thus does not degrade the predictions. For the same reason, a s-a-1 fault on this line will always cause the XOR gate to hash the address in a manner that simply permutes the FSMs. Therefore the original and new mappings are nearly *independent* and *isomorphic*. As a result, similar prediction accuracy (thereby almost no degradation) are obtained despite the existence of the fault.

The BHT (Shift Register): A SAF associated with line-index 4 fills the BHT with either all 0's or 1's, and, in effect, negates the usefulness of the history table. The resulting degradation in prediction accuracy is 13.23% and 14.52% for the s-a-0 and s-a-1 faults, respectively (not shown in **Figure 5** due to the scale of the y-axis). This demonstrates the usefulness of the BHT. When a fault is injected into the 9th flip-flop of the BHT (line-index 5), the degradation in prediction accuracy is only 9.26% for both s-a-0 and s-a-1, and 0.98% for both faults at line-index 6. Clearly, as one reduces the amount of history, the prediction accuracy also reduces. These faults are not isomorphic-equivalent.

XOR-Array: Line-index 10 refers to the 0^{th} output bit from the hashing function. When this line is either s-a-0 or s-a 1, the degradation is about 1.61%. The corresponding [s-a-0, s-a 1] degradations with respect to the inputs to this XOR gate, are [0.20%, 0.24%] for the instruction address as shown above and [0.59%, 0.62%] for the inputs from the BHT (Line-index 7). We believe that the degradation with respect to an error on the output of the XOR gate is larger than for an error on the corresponding instruction line because an error on the output can be caused by an error on either of the two gate inputs, hence there is more information lost. The reason that the degradation due to a fault at the output of the hashing function is much smaller than the corresponding fault in the BHT is because a fault in BHT affects the entire shift register, hence creating what is like a fault of multiplicity up to 18. For lines indices 11 and 12, the corresponding degradation figures are 0.55% and 0.98%, respectively, and are independent of the polarity of the fault. We believe the reason the degradation is not more uniform as one goes from one bit position to another is a function of the benchmark circuit and the addresses associated with branch instructions. **Table 4** shows the minimum and maximum degradation for stuck-at faults occurring at each input (from the BHT) and each output of the XOR-Array for five different benchmark programs.

Prediction Line: The degradation for *P* s-a-0 and s-a-1 are 42.8% and 48.5%, respectively; and for *A* s-a-0 and s-a-1 are 37.7% and 39.2%, respectively. These figures correspond to prediction accuracies ranging from 0.474 to 0.5734. Therefore, even for the most egregious faults, the predicted accuracies can be better than 50% in some cases.

While this result may seem surprising, at first a careful consideration of branch behavior at the application level will reveal the reasons for why even an egregious error, such as branch predictor output being stuck-at-1, still leads to 50% accuracy. Most applications for instance tend to user loop constructs (such as *for (i=0; i<n; ++i)* constructs in C) to repeatedly perform the same task. In the case of a *for* loop the branch is almost always taken. Hence, a stuck-at-1 fault produces the right result for many of these loopy branches. Similarly, constructs such as *do { ..} while (condition_true)* are also loop constructs where the branch is mostly not taken. Hence, a stuck-at-0 fault in this case correctly predicts the branch outcome.

In **Table 4** we show the degradation in prediction accuracy for faults on the lines *P* and *A* for five different benchmark programs. The faults on these lines lead to both the largest amount of degradation of all faults considered.

D. Cumulative Results

Cumulative relationships between the number of faults and their induced accuracy degradation for the *twolf* benchmark are derived using the data presented in **Table 4** and shown in **Figure 7**. The results show that 97.27% of the faults induce no degradation. Most of these faults are in the decoder and the FSM. 1.82% of the faults result in degradation between 0 and 1%. Finally, only 0.91% of the faults induce degradation between 1% and 48.49%.

In conclusion, most single stuck-at faults in the branch prediction unit cause very little degradation in prediction accuracy. In addition, as each of these faults is testable using the commonly used scan based test methodology, in current practice any chip with any of these faults will be discarded.

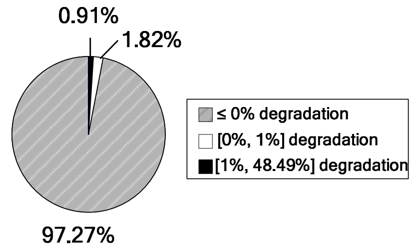

Figure 7: Percentage of faults inducing different levels of degradation in prediction accuracy for *twolf*

Clearly, many multiple faults also lead to small values of degradation, but this will not be quantified in this paper. Faults that create multiple 1's on the output of the decoder appear to result in potentially large values of degradation, but cannot exceed the case for *P* s-a-1.

The impact on yield enhancement is a function of the actual product being built. Ignoring all memories in a chip, assume that the area taken by the branch prediction circuitry represents the fraction *f* of the total chip area used by the logic circuitry. For example, say *f*=0.1. Consider all chips that have been identified to have stuck-at and/or bridge fault, and hence are to be discarded. If most of such chips have only a single fault, then about *f* (10% in our example) of these would likely have the fault lie in the branch prediction unit and hence might still be marketable, though using a different part number and lower price. Salvaging 10% of the chips that were to be discarded is **particularly significant**, since this additional yield is achieved at a nearly zero cost and hence directly augments the revenue.

3.4.2 Performance degradation

We now consider the impact of misprediction on performance. Assume a given benchmark program requires *K* clock cycles to run. When mispredictions occur, clock cycles are wasted in computing some results that are not needed as well as flushing the instruction and arithmetic pipelines so that they can process the correct stream of instructions. So now, as the misprediction rate increases due to a fault, it will take *K*'>*K* clock cycles before the program terminates. If the normal processing rate of the processor is $I\times10^8$ instructions per second, then when the misprediction rate increases, it will appear that the process is running at $(K/K')\times I\times 10^8$ instructions per second. We consider the quantity $P_d = (K'-K)/K$ to be the loss in performance due to a fault in the branch prediction circuitry. Clearly K' and thus P_d depends on the misprediction penalty, i.e., the number of cycles required to flush the pipeline due to branch misprediction. The penalty can be determined according to the pipeline depth of the target processor. For advanced processors that employ a deeper pipeline to improve the performance, the misprediction penalty is higher, and thus misprediction can usually result in larger P_d.

To determine P_d, we again use the SimpleScalar simulation tool, but now we carry out a different set of simulations to estimate the total number of clock cycles consumed due to a fault in the branch predictor. Because such simulations have higher run-time complexities than those used above to evaluate the prediction accuracy, and because faults causing similar accuracy degradation usually result in similar P_d, a smaller set of the faults are considered, each of which induces a significantly different accuracy degradation. Also, in our experiments three values of the misprediction penalty, namely 10, 20 and 30, are considered for the purpose of evaluating P_d for different penalties.

The relationship between the degraded prediction accuracy and the resulting P_d for *twolf* benchmark is illustrated in **Figures 8 and 9**. The results show that in the case where the penalty is 10, a fault inducing 10% degradation in prediction accuracy results in an P_d of 15%, and P_d goes up to 80% when 43% degradation is induced, which is clearly unacceptable. The results also show that all of the faults which induce less than 1% degradation result in less than 2% P_d (see **Figure 9**), which is acceptable for most applications.

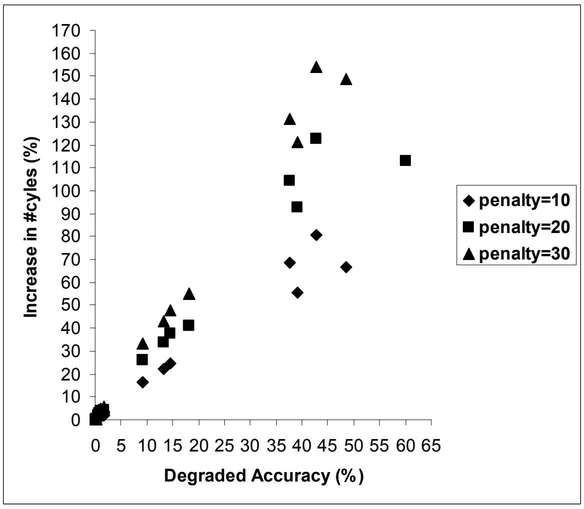

Figure 8: Relationship between degraded prediction accuracy and increase in the number of cycles for *twolf*

Also, as expected, P_d increases as the misprediction penalty increases. For example, a fault inducing 1% accuracy degradation results in P_d of 1.7%, 2.7% and 3.4% when the penalty is 10, 20 and 30, respectively. Therefore even 1% accuracy degradation might be unacceptable for a processor with a very deep pipeline architecture. Fortunately, our analysis results show that most of the faults in the target branch predictor results in no or negligible degradation, thereby leading to no or little increase in the number of cycles. As shown in **Figure 10**, 97.27% of the faults do not have any impact on the performance, while 1.82% of the faults induce increase between 0% and 1%. Only 0.91% will result in larger than 1% increase. Therefore, ***it is highly likely that a chip can work with almost the same performance even if it contains a defective branch prediction module***.

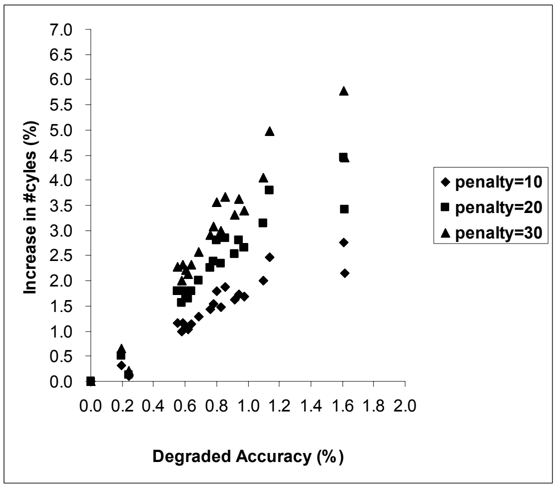

Figure 9: Zoom in of Figure 8 for increase in the number of cycles for degraded prediction accuracy ranging from 0% to 2%

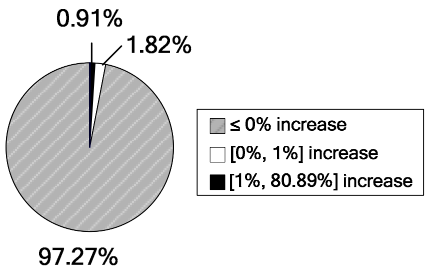

Figure 10: Percentage of faults inducing different levels of increase in the number of cycles for *twolf*

5. Conclusions

In this paper we have introduced a new fault category, namely performance degrading fault, which induces only performance degradation without causing any functional errors. This notion has also been experimentally validated via a detailed case study on the branch predictor in advanced processors. The case study shows that all the stuck-at faults in the branch predictor are performance degrading faults, and most faults have no or negligible impact on processor performance.

The proposed notion provides a new way to increase yield that has no, or in some cases little cost in additional hardware, and a small increase in test cost. It provides a completely new type of binning based on architecture-level performance that is totally different from the clock-frequency binning that is currently used by most microprocessor manufacturers.

6. Acknowledgements

This work was supported in part by the National Science Council of Taiwan (NSC) under Contract Nos. NSC097-2220-E-006-001 and NSC096-2917-I-006-120, and in part by the National Science Foundation of USA under Grant Nos. 0834798, 0428940 and 0720909. We also acknowledge the contribution of Dr. Ishwar Parulkar of Sun Microsystems who has followed our work on error-tolerance and suggested extending this work to processors, which are not allowed to produce erroneous outputs but may operate with some performance degradation.

7. References

[1] M. A. Breuer, S. K. Gupta, and T. M. Mak, "Defect and error-tolerance in the presence of massive numbers of defects," IEEE Design & Test of Computers 21(3): pages 216-227, 2004.

[2] M. A. Breuer and H. Zhu, "An illustrated methodology for analysis of error-tolerance," IEEE Design & Test of Computers 25(2): pages 168-177, 2008.

[3] H. Chong and A. Ortega, "Analysis and testing for error tolerant motion estimation," Int'l. Symp. on Defect and Fault Tolerance in VLSI Systems, pages 514-522, 2005.

[4] I. S. Chong and A. Ortega, "Hardware testing for error tolerant multimedia compression based on linear transforms," Int'l. Symp. on Defect and Fault Tolerance in VLSI Systems, pages 523-521, 2005.

[5] A. Agarwal, B. C. Paul, H. Mahmoodi, A. Datta and K. Roy, "A process-tolerant cache architecture for improved yield in nanoscale technologies," IEEE Trans. on VLSI Systems, 13(1), pages 27-38, 2005.

[6] P. P. Shirvani and E. J. McCluskey, "PADded cache: a new fault-tolerance technique for cache memories," VLSI Test Symp., pages 440-445, 1999.

[7] S. Almukhaizim, T. Verdel and Y. Markris, "Cost-effective graceful degradation in speculative processor subsystems: the branch prediction case," Int'l. Conf. on Computer Design, pages 194-197, 2003.

[8] J. Blome, S. Mahlke, D. Bradley, K. Flautner, "A microarchitectural analysis of soft error propagation in a production-level embedded microprocessor," in the Workshop on Architectural Reliability (WAR), 2005.

[9] Http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf.

[10] S. McFarling, "Combining branch predictors," WRL Technical Note TN-36, Digital Equipment Corporation, 1993.

[11] SPEC2000 website, www.spec.org/osg/cpu2000/.

[12] T. Austin, E. Larson and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling," IEEE Computer, 35(2): pages 59-67, 2002.

Table 4: Degradation due to faults in different modules for five benchmarks

Module	Line type	Total # faults	Line index	Degradation	*twolf*		*mcf*		*vpr*		*gap*		*gzip*	
					SA0	SA1	SA0	SA1	SA0	SA1	SA0	SA1	SA0	SA1
Decoder	A	72	[A_1, A_6]	Min.	0.0	0.4	0.0	0.2	0.0	0.0	0.0	0.1	0.0	0.0
				Max.	0.0	1.2	0.0	0.4	0.0	0.4	0.0	0.4	0.0	0.1
	B	18,432	[B_1, B_6]	Min.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
				Max.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	C	2,048	[C_1, C_6]	Min.	11.1	0.0	2.9	0.0	2.7	0.0	0.0	5.7	0.0	0.1
				Max.	15.0	0.0	7.9	0.0	11.0	0.0	0.0	9.6	0.0	4.7
	D	1,048,576	[D_1, D_6]	Min.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
				Max.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	E	524,288	[E_1, E_6] ([13, 18])	Min.	0.0	18.3	0.0	19.9	0.0	15.4	0.0	20.4	0.0	26.8
				Max.	0.0	18.3	0.0	19.9	0.0	15.4	0.0	20.4	0.0	26.8
FSMs	MSB	13,634,188	[19, 24]	Min.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
				Max.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	LSB	13,634,188	[25, 30]	Min.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
				Max.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Branch Inst. Addr.	N/A	36	[1, 3]	Min.	-0.1	-0.1	-0.1	-0.1	0.0	0.0	0.0	0.0	0.0	0.0
				Max.	0.2	0.2	0.2	0.2	0.4	0.4	0.3	0.3	0.2	0.2
BHT	N/A	36	[4, 6]	Min.	1.0	1.0	0.2	0.2	0.4	0.4	0.0	0.0	0.0	0.0
				Max.	13.2	14.5	7.7	7.7	4.8	4.8	4.5	4.5	6.4	6.4
XOR	Input	36	[7, 9]	Min.	0.0	0.0	-0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
				Max.	1.0	1.0	0.5	0.5	0.5	0.5	0.1	0.1	0.0	0.0
	Output	36	[10, 12]	Min.	0.6	0.6	0.2	0.2	0.0	0.0	0.2	0.2	0.0	0.0
				Max.	1.6	1.6	0.8	0.8	0.5	0.5	0.6	0.6	0.2	0.2
P	N/A	2	31	Min.	42.8	48.5	51.4	41.2	40.2	53.2	57.7	39.5	60.5	33.0
				Max.	42.8	48.5	51.4	41.2	40.2	53.2	57.7	39.5	60.5	33.0
A	N/A	2	32	Min.	37.7	39.2	49.0	37.8	34.1	48.8	42.4	31.0	60.4	31.2
				Max.	37.7	39.2	49.0	37.8	34.1	48.8	42.4	31.0	60.4	31.2

Note: **[*i*, *j*]** represents the lines indexed by from *i* to *j*