Reliability Aware Exceptions for Software Directed Fault Handling

Waleed Dweik, Murali Annavaram, Michel Dubois*
Department of Electrical Engineering
University of Southern California
Los Angeles, USA
{dweik, annavara}@usc.edu, dubois@paris.usc.edu

Abstract—Today reliability emerges as a first order design constraint. Faults encountered in a chip can be classified into three categories: transient, intermittent and permanent. Fault classification allows a chip designer to provide the appropriate corrective action for each fault type. However, fault classification and correction are expensive mechanisms to implement in hardware. In spite of their criticality faults are still relatively rare; hence classification and recovery mechanisms should be very low cost. In this paper, we present a new class of exceptions called Reliability Aware Exceptions (RAE). RAE is a software mechanism with minimal hardware cost which provides the ability to classify the cause of a fault to one of the three categories. Fault detection is done in hardware and once a fault is detected the hardware raises an exception. Exceptions are handled by software where classification and the resulting fault handling algorithms run as specialized exception handling routines. The exception handlers are equipped with the ability to manipulate microarchitectural blocks to recover from all three categories. For a transient fault recovery RAE leverages the existing roll back mechanism provided for branch misprediction to flush the pipeline and re-execute starting from the faulting instruction. For an intermittent faults RAE exploits available redundancy in the microarchitecture to de-configure the faulty units temporarily, while for hard faults the unit can be permanently de-configured. We present a detailed RAE implementation and then evaluate the effectiveness of the RAE approach to protect Reorder Buffer (ROB). Our results show that ROB's FIT (Failures in Time) rate can be decreased by a factor of 2 using the RAE mechanism.

Keywords-Field De-configurable Unit (FDU); Failure in Time (FIT); Reliability; Duty Cycle; Storage Structures

Submission Category- DCCS: The Dependable Computing and Communications Symposium

Word Count- 10681

I. INTRODUCTION

Technology scaling has led to variations in device characteristics resulting in a range of susceptibilities. Faults encountered in a chip can be classified into three categories: transient, intermittent and permanent. The most prominent faults are: transient faults due to particle strikes, time dependent dioxide breakdown (TDDB), electro-migration (EM) and, Negative Bias Temperature Instability (NBTI) [14][13][12][16]. As the gate oxide thickness is decreasing with technology node, break down in oxide layer leads to TDDB induced faults. The shrinking width of interconnects and vias accelerates EM wear out effects. NBTI affects PMOS devices that have an extended period of negative bias stress. These are dominant factors limiting chip lifetime [22]. Each of these fault categories has different severity and different probability of occurrence depending on the kind of stress condition exerted on a device. Correspondingly, the effectiveness of error recovery solutions can be improved if the recovery mechanisms are cognizant of the fault type. Another salient characteristic of these faults is that they occur slowly over time and in some cases the device can recover from faults when the stress condition is removed, such as NBTI. Hence, it is necessary to develop very low cost solutions to classify and handle various fault types.

In this paper we propose Reliability Aware Exceptions (RAE), a special class of exceptions that enable software directed fault handling, which can be used in conjunction with low-cost hardware error detection mechanism to improve chip life time. The novelty of the RAE mechanism is the ability to classify the fault to one of the three main categories: transient (soft), intermittent, and permanent (hard) faults for different field de-configurable units (FDUs). The occurrence of an error is associated with the instruction that first encounters or triggers a fault. Once a fault is detected, the ROB entry allocated to the instruction that exercised the fault is updated with the exact location of the faulty FDU (ALU, Multiplier, Entries of memory structures ...etc). The faulty FDU is temporarily disabled from further use by using mechanisms described in [3] and [4]. Just as in a traditional exception handling mechanism an exception is triggered only when the faulting instruction reaches the top of re-order buffer for commit. Once an exception is triggered the software handlers use the FDU location information stored in the ROB to identify the faulty FDU and mines the historical fault logs to determine the most probable cause of the failure. This probably cause essentially categorizes the fault. When a fault is categorized, RAE helps in choosing the most appropriate handler that gives best performance and cost trade off according to the fault type. For instance, when a specific FDU exercises an intermittent fault, we de-configure (turn OFF) the faulty FDU temporarily. This is particularly useful for NBTI induced faults, as the de-configuration will remove the stress condition on the faulty FDU and partially recover to the original device state. For EM or TDDB induced faults, no recovery is possible. However, RAE increases the mean time to failure (MTTF) by reducing the stress conditions.

This idea of disabling the faulty component is well utilized in computing industry. It has been used to map out

bad sectors in disks [3], disable functional blocks that are deemed to have encountered hard failures [10][15]. To our knowledge, this work is the first to classify faults and distinguish between transient, intermittent and permanent faults. We enable various levels of de-configuration of FDUs to mitigate the impact of faults.

While reading the rest of the paper it is important to distinguish between the two terms: disable and de-configure. Disabling a specific FDU means to prevent the FDU from further use, but the FDU is not turned OFF. De-configuring an FDU means to turn OFF the FDU by disconnecting its voltage source. For non-storage structures (ALU, Multiplier/Divider) the word FDU means the entire structure. For storage structures (Queues, Caches), FDU means the building block of that structure (Queue Entry, Cache Line).

The rest of the paper is as follows. We explore RAE design details, including the modifications necessary at the microarchitecture level in section 2. Section 3 presents our experimental infrastructure, fault injection approach and simulation results. Section 4 reviews related work and we conclude in section 5.

II. BUILDING BLOCKS FOR RAE DESIGN

In this section we describe the major design components of the RAE mechanism. The RAE mechanism is composed of three main components: fault detection, temporarily disabling the faulty FDU from further use, and using RAE handler to classify the fault and choose recovery action.

Fig. 1 gives a high level overview of the RAE implementation. In particular, fig. 1 focuses on how RAE can be used to protect the ROB. The ROB is protected with a parity checker for error detection. When a fault is detected, the RAE bit of that entry is set to indicate that there is a reliability aware exception associated with that entry. In addition, the location field "Loc" in faulty ROB entry will be updated with the FDU identification number (FDU_ID) which in this case is the entry number itself. Then the faulty ROB entry is disabled by setting its "Busy" bit to 1. The RAE mechanism does not trigger any exception until the faulty ROB entry becomes the top (oldest instruction). At that moment the processor is flushed and the RAE handler starts execution. Handler will categorize the fault type based on fault logs stored in the system. After the handler completes, based on fault type we take a corrective action, which may be as simple as restarting the execution from the faulty instruction or could be as complex as de-configuring the ROB entry permanently.

A. *Fault Detection*

The first step in RAE mechanism is to detect the faults as soon as they occur. As an instruction passes through various pipeline stages it spends a significant fraction of time waiting in various storage structures, such as instruction fetch queue (IFQ), Reservation Station (RS), ROB, load/store queue (LSQ). It may also access other storage structures such branch history table (BHT), register files, and register map table. Hence in this paper, we

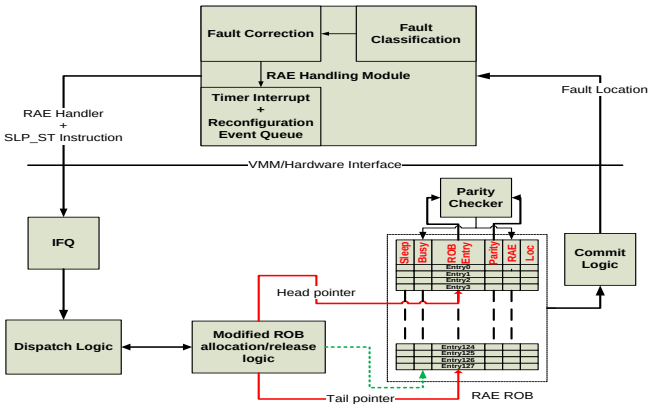

Figure 1. RAE Implementation for ROB

primarily focus on the implementation of RAE mechanism for protecting against failures in storage structures. To detect a fault in these structures, we propose to use single parity bit for every entry in each of the processor storage structures; since parity has negligible area overhead. We assume that no more than one fault occurs in any entry that is protected by parity. Note that fault detection is a not a novel component of our work; we simply use parity to detect an error in storage structures. To detect timing and computational errors in non storage structures, such as ALUs, one could use partial redundant execution using look-ahead circuits proposed in [8] or any of techniques proposed in [20] when applying RAE to them. RAE is a flexible mechanism and it does not matter which detection mechanism one chooses provided that it detects faults and flags the faulty instructions.

When a fault is detected in a specific FDU while executing an instruction, the first thing to do is to set the RAE bit in the instruction's ROB entry to 1. This bit indicates that there is a reliability exception associated with this instruction and must be handled when the instruction reaches the top of the ROB. The location of the fault (FDU_ID) must be stored as well so that it can be used later in RAE handling stage to help classifying the fault and de-configuring the faulty FDU when needed. As shown in fig. 1, the ROB is augmented such that each ROB entry is extended with a new field called "Loc" to the FDU_ID of the faulty FDU. As mentioned earlier FDU_ID can be simply the ID of the entire structure such as ALU, or it can be finer grain identification information such as the entry number within a storage structure. Notice that the FDU_ID can be the ROB entry itself.

Consider the case of protecting ROB in an out-of-order processor without a physical register file, where the ROB stores the instruction result and updates architecture register file during commit. Each ROB entry is first allocated and updated with destination register tag and control information during the instruction dispatch stage. The ROB entry is then updated once again when the instruction completes execution and writes the result to the ROB entry. At the time of commit the ROB value is read and stored in the architecture register. Hence, every instruction accesses the ROB at least three times in the pipeline. Whenever the entry

is updated with new data, the new parity is computed and saved. Every time the ROB entry is read parity is computed and compared with the saved parity. If there is a match no fault is detected, otherwise a fault is detected and the FDU_ID (ROB ID + entry number) is stored in the location field of the faulty ROB entry. In this simple illustration ROB entry is continuously protected with parity from the time it is allocated to the time it is released. The area overhead of parity computation and comparison hardware (represented by the parity checker block in fig. 1) is negligible and parity computation can also be done off the critical path in most cases.

B. Disabling the Faulty FDU

Once a fault is detected the faulty FDU is also immediately disabled. Disabling the faulty FDU is necessary to ensure that once a fault is detected no later instructions should access the same FDU until the exception handler deals with the fault. Immediately disabling the faulty FDU also guarantees that when the RAE handler executes its instruction they will not exercise the same fault and hence guarantee forward progress. Hence; when a fault is detected, the detection hardware will immediately disable the faulty FDU.

Microarchitectural storage structures can be categorized according to their access pattern into two groups: circular buffer array structures (ROB, IFQ, and LSQ) and tabular (i.e. directly addressed) array structures (BHT, RS, and register file). In [3], Bower et al. propose to add one level of indirection to the allocation/release logic of circular buffer array structures in order to disable faulty entries. We propose to use the same technique; each entry of every buffer array structure is augmented with a busy bit as shown in fig. 1. The faulty entry is disabled by simply setting its busy bit to 1. The allocation/release logic of the buffer array structure must also be modified by adding one level of indirection to check the busy bit before allocating a new entry to see if it is faulty or not. If the entry is faulty, the allocation/release logic will skip that entry and check the next for allocation. The selection of which entry is to be allocated next happens off the critical path and hence does not affect the read/write access times.

On the other hand, tabular array structures are randomly accessed using an address. The allocation/release logic of tabular array structures keeps track of which entries are in-use and which are free. Hence, we can simply disable faulty entries in tabular array structures by marking them in the allocation/release logic as in-use [4]. Unlike circular buffer array structures, there is no need to add a busy bit and a level of indirection to the allocation/release logic for tabular array structures. The same approach can be used when implementing RAE for non-storage structures by simply marking the faulty structure (ALU, Multiplier) as in-use.

Fig. 1 shows how ROB entries can be disabled. ROB entries are augmented with busy bits that are assumed to be fault free. In addition, the ROB allocation/release logic is modified in order to check the busy bit before allocating a new entry or before deciding which entry is the new top of the ROB. Once a fault is detected in a specific ROB entry, the fault detection hardware (parity checker) is responsible for setting the busy bit of that ROB entry to 1. When the fault is handled, none of the handler's instructions will be allocated to the faulty ROB entry because it is disabled and hence will be skipped by the allocation/release ROB logic.

C. RAE Handler: Fault Classification

As is the case with traditional exceptions, RAE stays silent until the faulty instruction is ready to commit. By the time the instruction reaches the top of the ROB the faulty FDU is disabled and all previous instructions have committed their result. Hence, the processor pipeline is flushed leveraging the same flush mechanism used for branch misprediction and exception handling. The program counter register (PC) is loaded with the address of the first instruction of the RAE handler.

The RAE handler consists of two main parts: the first part takes care of fault classification and the second part is responsible for choosing the most appropriate recovery action according to the class of the fault. RAE classifies a fault into 3 categories: transient, intermittent, and permanent. Classification relies on mining the fault history logs. Fault history logs are disk-based files that record every RAE event in a database. Each entry in the database contains the following information: Timestamp (when the last RAE was raised), FDU_ID (where the error was detected), Last Fault Classification (what the handler classified the error the last time it was encountered). Once RAE is raised the fault handler access this disk based fault log database and fetches all database records associated with the FDU_ID by using the ID as the search key in the database.

Accessing disk based databases for fault classification and handling is a slow process. But under the assumption that faults are still very rare events we can afford to handle worst case scenarios using slow software handlers. What is critical is to avoid Silent Data Corruption (SDC) errors where an error goes undetected. By using just error detection hardware we can translate SDCs into Detected but Unrecoverable Errors (DUEs). Once a DUE is encountered RAE handlers prevent machine from crashing, to the extent possible, thereby increasing the MTTF. RAE handlers may eventually run out of options to take any corrective action leading to a machine crash. Furthermore, RAE handlers can also provide early warning signs to system administrators to replace failing components where possible or the entire system. The RAE handlers that classify and correct errors thus form the crux of our approach to improve MTTF.

Since RAE handlers are written in software these are highly flexible and a system manufacturer can change the handlers based on long term reliability data obtained from the fault logs. In this paper we use a classification algorithm that relies on counters and probability of occurrence of a fault type to categorize the fault. In particular, our classification algorithm uses a constant: $C = (0.95)*(1\div(SER/sec))$, where SER stands for soft error rate. Typically SER is measured as probability of a transient fault in a single clock cycle. By knowing the clock frequency we

translate SER into probability of a transient fault in one second (SER/sec). C is thus 95% of the time interval (in seconds) between two transient faults.

We will now describe our fault classification algorithm using the example of ROB structure shown in fig. 1. When the system is first built the fault log database is empty. Assume during system operation a fault detected on entry#4 in the ROB at time T1. Immediately, the hardware disables entry#4 by setting the busy bit to 1 and the FDU_ID (ROB_4) is stored in the "loc" field of entry#4. When the first RAE is raised (when entry#4 becomes top of the ROB) the hardware takes two steps: First, the pipeline is flushed by the hardware treating this fault as a branch misprediction. Second, the RAE handler is invoked. The handler then searches the fault log database using ROB_4 as the search key. Since the log is empty no records are returned. The handler simply classifies the current fault as transient. The handler also enters the RAE record in the database. The record will contain the information: <ROB_4, T1, Transient, Transient_Action>. We will describe the Transient_Action taken by the handler in the next section.

Now assume that at a future time T2 a new fault is detected in ROB entry#4. As with the first fault, entry#4 is disabled and FDU_ID is stored. Once entry#4 reaches top of the ROB, the hardware flushes the processor and invokes RAE handler. The handler accesses the database with ROB_4 as the key. The database this time returns one record indicating that one RAE has been recorded for ROB_4. If the last fault type is transient, the handler computes the time interval between T2 and T1 in seconds. If the interval (T2-T1) is greater than or equal to C, then the current fault type is again categorized as transient. The assumption is that if the last failure occurred at least after C seconds then most probably the new fault is also transient. If the interval is less than C then the handler now categorizes this fault as intermittent. Depending on the classification the handler inserts a new fault record in the database with the information <ROB_4, T2, Transient, Transient_Action> or <ROB_4, T2, Intermittent, Intermittent_Action>. We will describe the Intermittent_Action taken by the handler in the next section.

Finally, assume at time T3 a fault was detected on ROB_4. Then the handler will access the database and gets two fault records, corresponding to T1 and T2 events. It then takes the most recent event (T2) and computes the interval between T3 and T2. If the interval is >= C, the handler categories the event at T3 as transient again and enters <ROB_4, T3, Transient, Transient_Action > record in the database. If interval < C then the handler counts the number of times ROB_4 was classified as Intermittent. If the count is less than or equal *Intermittent_Threshould*, then the fault event at T3 is still treated as intermittent. The handler enters <ROB_4, T3, Intermittent, Intermittent_Action> record in the database. If count > *Intermittent_Threshold* then the handler categorizes the fault as permanent. It then enters <ROB_4, T3, Permanent, Permanent_Action> in the database. The value of *Intermittent_Threshold* is specified by the system designer taking into consideration process technology node and expected operating environment of the system in field. It can also be arbitrarily set to a large value. In the worst case the handler will erroneously categorize a permanent fault as intermittent fault thereby causing some performance degradation and power overhead, as we will show later. We will describe the Permanent_Action taken by the handler in the next section.

D. RAE Handler: Fault Correction

Now we know the class of the fault, the appropriate corrective action can be chosen to mitigate fault effects. There are 3 actions:

1) Transient_Action: Transient faults persist only for a single cycle; hence, no corrective action other than restarting from the faulting instruction is needed. However, the hardware has already disabled the faulty FDU. In our example scenario, the hardware disabled ROB_4 by setting the busy bit to 1. The RAE handler directs the hardware to reset the busy bit to 0 in case of circular buffer array structures or mark faulty FDU as free in case of tabular array structures and non storage structure.

2) Intermittent_Action: Intermittent faults mainly include TDDB, EM, and NBTI faults. The recovery action is to keep the faulty FDU disabled for longer period of time and when possible it is best to even de-configure the FDU. Temporal de-configuration (turning OFF) of the faulty FDU is very helpful especially in the case of NBTI. By de-configuring the faulty FDU, we remove the negative bias stress applied to PMOS transistors inside that FDU. This helps the PMOS transistors to almost recover to their original state and threshold voltage. To our knowledge there is no easy recovery possible for TDDB and EM faults for any arbitrary FDU; however, temporal de-configuration increases the MTTF by reducing the stress conditions. In our approach the length of the de-configuration period follows exponential distribution based on the number of consecutive occurrences of the intermittent fault. For instance when the fault is first classified as intermittent fault the FDU is turned OFF for 1 million cycles. On consecutive intermittent faults the FDU is turned OFF for 2, 4, 8 … million cycles until the number of faults reach the *Intermittent_Threshold* value. Since the handler is implemented in software, the de-configuration can even be dynamically controlled over the FDU lifetime. During early life stage of the systems, the de-configuration period can be of order of 1000's cycles since system may recover quickly in the initial stages of its operational life. In later stages of system lifetime, it will be useful to increase the de-configuration period to billions of cycles for better recovery. Once the de-configuration period of the faulty FDU expires, the faulty FDU is reconfigured into the normal system operation as we will explain shortly.

3) Permanent_Action: Permanent faults persist forever and whenever the faulty FDU is used an error will occur except in cases where the fault is logically masked. Since masking effects cannot be easily measured at runtime, for permanent faults we choose to keep the faulty FDU disabled and permanently turn it OFF. The main advantage is to avoid large number of flushes and re-executions for

each time the faulty FDU is used during computation. Another side benefit is the power savings due to reduced number of flushes.
After the handler completes its execution, we roll back and start the execution from the instruction that exercised the fault.

E. FDU De-configuration Approach

Next we describe our de-configuration mechanism. In our description, we mainly focus on NBTI faults which greatly benefit from the de-configuration. Most storage structures, whether they are circular buffers or tabular arrays are built using traditional 6-Transistor SRAM cells. Each cell has two back-to-back inverters and two pass transistors connected to the same word line. Only the two PMOS transistors in every cell are susceptible to NBTI faults. One important observation is that whether the value stored in the cell is logic 0 or logic 1, one of the two PMOS transistors will be stressed (negative gate-source voltage) while the other will be recovering (zero gate-source voltage). The problem occurs when the SRAM cell maintains its value for a long time without flipping; in that case one of the PMOS transistors will be stressed for a long time which will cause its threshold voltage (V_T) to shift and may introduce a fault when the cell is read. The temporal de-configuration helps us to overcome this problem by making sure both PMOS transistors are recovering (turned OFF) during this period.

The de-configuration is achieved through the use of sleep transistor [17][18]. We chose to use header sleep transistor implementation; which will isolate the voltage source from the source nodes of PMOS transistors when the sleep transistor is OFF. In addition to the sleep transistor, we also add a weak NMOS transistor which helps in discharging the source nodes of the two PMOS transistors to zero during the de-configuration period. Applying zero voltage at the source nodes of transistors will ensure that both transistors will be in the recovery mode during the de-configuration period.

De-configuration granularity is a design choice. For instance, in our example in fig. 1 we chose to de-configure any single ROB entry. Hence, we implemented sleep transistors at the same granularity by having a single sleep PMOS transistor and a single discharging NMOS transistor for each ROB entry. In addition, ROB entry is augmented with a sleep bit (SRAM cell). The sleep bit of each entry drives both sleep and discharging transistors of that entry. Initially, all sleep bits in all storage structures are initialized to "0", which means that the sleep transistor is ON (connected to the voltage source) and the discharging transistor is OFF. When a fault is classified as intermittent or permanent, RAE handler sets the sleep bit to 1 thereby cutting off power supply to the FDU.

We provide a single new instruction called SLP_SET that is used by the RAE handler to de-configure or reconfigure the FDU. SLP_SET takes FDU_ID and fault type as source operands. This new instruction does not have to be made available to high-level software. Only the fault handler uses this instruction and the system designers write this handler.

For transient faults, RAE handler needs to enable the FDU by directing the hardware to reset the busy bit in case of circular buffer array structures or mark FDU free in case of tabular array structures and non storage structure. RAE handler uses the SLP_SET instruction to achieve that. For intermittent or permanent faults, RAE handler uses SLP_SET instruction to set the sleep bit of the FDU to 1. If fault type is permanent there is no need to do anything else as the FDU will be turned OFF forever. If the fault type is intermittent, RAE needs to schedule an event to reconfigure the FDU when the de-configuration period expires. RAE keeps a queue of reconfiguration events for all FDUs that are currently de-configured because of an intermittent fault. The queue is kept in sorted order with the reconfiguration event that will occur earliest is at the head of the queue. RAE handler uses a timer interrupt which iteratively polls the register that contains the wall clock to check if the reconfiguration event at the head of the queue is due. When the top reconfiguration event is due, RAE uses SLP_SET instruction to reconfigure and enable the FDU by directing the hardware to set the sleep and busy bits to 0.

Fig. 1 shows RAE handler module with 3 sub modules (Fault Classification, Fault Correction, and Timer Interrupt with Reconfiguration Event Queue). Each ROB entry is augmented with 4 more bits (parity, RAE, busy, sleep) and 16-bit FDU_ID field (loc). All the other circular buffer array structures, such as LSQ and IFQ need to be augmented with only 3 bits (parity, busy, sleep). They use parity to detect faults, use busy bit to prevent any later instruction from using the FDU, and use the sleep bit to completely turn OFF the FDU. Since the occurrence of a fault is always associated with the instruction that exercises the fault and every in flight instruction is allocated to a specific ROB entry, only ROB needs to be augmented with RAE bit (to indicate that there is a reliability fault associated with this instruction) and loc field (to hold the FDU_ID of the fault). On the other hand, tabular array structures are augmented with 2 bits (parity and sleep) because enabling/disabling the entries of these structures is accomplished by marking them as being free or in-use. In our simulated machine, we have 128-entry ROB, 128-entry RS, 64-entry LSQ, and 4-entry IFQ. The total area overhead of implementing RAE mechanism to protect these storage structures is (128*20) + (68*3) + (128*2) = 3020 bit which is negligible compared to reliability and performance improvement achieved as we will show in section III part C.

The use of exceptions to handle complex functions in software is a well established design tradition in most modern systems to deal with rare events such as page faults, TLB misses. In a sense, we are extending the exception classes to include reliability aware exceptions to handle faults which are also rare events. The main advantage of using software functions in such complex events is that they provide full resource exploitation and can be easily modified and maintained in the field. On the other hand, it is a known fact that software functions are much slower than hardware functions and hence can incur higher performance degradation. However, the rarity of fault events allows us to benefit from the software flexibility to handle reliability

faults while incurring minimum performance degradation compared to a machine with hardware based recovery mechanism.

III. EVALUATION METHODOLOGY AND RESULTS

Now that we showed how RAE works, for the purpose of demonstrating the impact of RAE classification and fault-specific corrective actions we focus on how to protect one storage structure, namely the ROB. In this section, we will describe our experimental infrastructure, fault injection approach, and simulation results from 8 benchmarks for base machine (without RAE support) and RAE machine.

A. Experimental Setup

RAE mechanism is evaluated using execution-driven simulations with a detailed processor model. The processor model simulated is a 4 wide Out-of-Order processor with 128-entry ROB, 128-entry RS, 64-entry LSQ, 16KB L1 direct-mapped data cache, 4KB direct-mapped L1 instruction cache, 1MB 8-way associative L2 unified cache, 4 ALUs, 1 integer Multiplier/Divider, 1 floating point ALU, and 1 floating point Multiplier/Divider. The processor model stores the speculative results from instruction execution in the ROB rather than in the physical register file. When an instruction is committed these results are then written to an architected register file. For our simulations, we modified Simplescalar simulator [7] by integrating Wattch [6] and HotSpot [9] to measure temperature at any given block in a processor's floor plan. We focus our evaluations on the ROB which is a large on-chip SRAM storage structure. We expect qualitatively similar results to hold when the RAE mechanism is applied to protect other storage structures like IFQ, LSQ, and RS. Every ROB entry in our implementation is 40 bits wide: 32 bit data field for storing the speculative result, 5 bit address field for the architectural destination register address, and 3 control bits that track when instruction execution is completed and instruction type.

B. Fault Injection

Our simulation experiments involve injecting faults of the 3 main categories (transient, intermittent, and permanent) in the ROB of both base and RAE machines and then compare their performance measured in instruction per cycle (IPC) and reliability measured in FIT. The challenge is to inject faults that mimic the physical phenomena of the three fault categories, transient, intermittent and permanent. In other words, we need a probability distribution of the 3 main fault types in order to know the likelihood of the injected fault being of a certain type.

In [19], Shin et al presented an architectural-level lifetime reliability modeling framework. The framework is based on a new concept called the FIT of reference circuit (FORC) which is relatively easy to model while allowing us to compute the FIT rate due to TDDB, EM, and NBTI fault mechanisms effectively. We propose to use these models to compute the FIT value for TDDB, EM, and NBTI failure mechanisms and then use those FIT values to compute the relative likelihood of occurrence of a given fault type. The relative likelihood computed is then used to generate the probability distribution of three categories.

In our simulation, we treat TDDB and EM faults as permanent faults as there is no easy recovery process for such faults. NBTI faults are recoverable; hence, they count for all intermittent faults in our simulation. Soft error strikes count for transient faults. The probability of each fault category is computed as follows: $P_{Transient} = FIT_{Transient}/FIT_{Total}$, $P_{Intermittent} = FIT_{NBTI}/FIT_{Total}$ and $P_{Permanent} = (FIT_{TDDB} + FIT_{EM})/FIT_{Total}$. Where $FIT_{Total} = FIT_{Transient} + FIT_{NBTI} + FIT_{TDDB} + FIT_{EM}$. $FIT_{Transient}$ is the number of transient faults in billion hours and it is constant throughout the simulations. According to ITRS 2007 report [17], one Mega Byte of SRAM in 65nm process technology is projected to have transient FIT rate of 1150, also referred to as intrinsic FIT. Using the same projection for our ROB structure with 5120 bits (128 x 40), the ROB transient FIT value ($FIT_{Transient\text{-}ROB}$) is 1. So, one transient fault is likely to occur in ROB every one billion hours.

Next, we present the FORC and FIT equations which we used for TDDB, NBTI, and EM faults respectively. The FORC and FIT equations of TDDB faults are derived in [19] and are show below:

$$\mathrm{FORC_{TDDB}} = \frac{10^9}{A_{TDDB}} \times V_{dd}^{a-bT} \times e^{-\frac{X+\frac{Y}{T}+ZT}{kT}} \quad (1)$$

$$\mathrm{FIT_{TDDB\_ROB}} = \sum_{FET} \mathrm{duty\_cycle} \times \mathrm{FORC_{TDDB}} \times (\#\ \mathrm{of\ ED}) \quad (2)$$

where A_{TDDB}, a, b, X, Y and Z are fitting parameters derived empirically [21]. T is the temperature of the FDU in Kelvin and k is Boltzmann's constant. So, first we compute the failure in time value of the reference circuit for TDDB faults ($FORC_{TDDB}$) using (1). After that, we use (2) to compute the FIT_{TDDB} value of the ROB. Each SRAM cell has 6 field effect transistors (FET); we find the sum across the 6 FETs. The duty cycle and the number of effective devices (ED) for all the 6 FETs are given in [19].

The FORC and FIT equations of NBTI faults are derived in [19] and are show below:

$$\mathrm{FORC_{NBTI}} = 10^9 \times \left(\frac{K}{\Delta V_{T\_ref}}\right)^{\frac{1}{n}} \quad (3)$$

$$\mathrm{FIT_{NBTI\_per\_pFET}} = 10^9 \times \left(\frac{\Delta V_{T\_ref}}{\Delta V_c}\right)^{\frac{1}{n}} \times \left(\frac{\mathrm{FORC_{NBTI}}}{10^9}\right)^{\frac{0.25}{n}} \quad (4)$$

where ΔV_{T_ref} is the maximum allowable shift in V_T of the reference circuit, ΔV_c is the maximum allowable shift in V_T in the circuit that contains the pFET device (ROB), and n is a function of the duty cycle. The equation to compute parameter K is given in [19]. First, we compute the failure in time (FIT) value of the reference circuit for NBTI faults ($FORC_{NBTI}$) using (3). Second we use (4) to compute the FIT value of a single PMOS device for NBTI faults ($FIT_{NBTI\text{-}per\text{-}PFET}$). Finally, we multiply the result of (4) by the total number of PMOS devices in ROB (128*40*2 = 10240)

to get the FIT value of the entire ROB for NBTI faults ($FIT_{NBTI\text{-}ROB}$).

In [2], Black presented a model to calculate MTTF due to EM failure mechanism. Based on Black's equation, Shin et al proposed the following equations to compute the FORC and FIT of EM faults [19]:

$$FORC_{EM} = \frac{10^9}{A_{EM}} \times \left(\frac{C_{ref} \times V_{dd}}{t}\right)^n \times e^{\frac{-E_{\alpha-EM}}{kT}} \quad (5)$$

$$FIT_{EM\_ROB} = N_{cells} \times N_{ports} \times \left(\frac{C_{bitline}}{C_{ref} \times N_{rows} \times N_{ports}}\right)^n \times \left[\begin{array}{l}\left(N_{reads} \times P_1 + \frac{1}{2\gamma} \times N_{writes} \times P_{flip}\right)^n \\ + \left(N_{reads} \times P_0 + \frac{1}{2\gamma} \times N_{writes} \times P_{flip}\right)^n\end{array}\right] \times FORC_{EM} \quad (6)$$

where A_{EM} and n are empirical constants, E_{α_EM} is the activation energy for EM, k is Boltzmann's constant, T is absolute temperature in Kelvin, t is the clock period, C_{ref} is capacitance of reference circuit, $C_{bitline}$ is capacitance of the bitline in the array structure under test (ROB), N_{cells} is number of SRAM cells in the ROB (5120 = 128*40), N_{ports} is number of read/write ports in our ROB, N_{rows} is total number of ROB entries (128), N_{reads} and N_{writes} is total number of read and write operations from/to the ROB, and γ is the duty cycle to pull up transistors in the cells. P_0 and P_1 are the probabilities of having logic 0 or logic 1 in a cell, and P_{flip} is the probability of flipping a bit. Just as before we need to compute the failure in time value of the reference circuit for EM faults ($FORC_{EM}$) using (5) and then use that to compute the FIT value of ROB for EM faults ($FIT_{EM\text{-}ROB}$) using (6).

As can be seen from the above description, TDDB, NBTI, and EM faults depend on the operational condition. In our fault injection model we mimic the physical phenomena by accurately measuring the various environmental conditions (such as temperature) and stress conditions (such as P_0, P_1) to measure the FIT rate of each fault type. We then compute the relative probability of occurrence of each fault type during runtime. We keep track of the ROB average temperature using Hotspot [9]. Hotspot requires a chip floor plan, so we created a microprocessor chip floor plan similar to Intel P4 floor plan given in [11] using QUILT (Quick Utility for Integrated circuit Layout and Temperature modeling) [5]. Stress conditions, such as P_0, P_1, and P_{flip} are measured using performance counters added to the simulator. The above activity and thermal information is used to calculate $FIT_{TDDB\text{-}ROB}$, $FIT_{NBTI\text{-}ROB}$, and $FIT_{EM\text{-}ROB}$ dynamically every 5 million cycles using (1), (2), (3), (4), (5), and (6). $FIT_{Transient\text{-}ROB}$ is constant (= 1) during the entire simulation. As mentioned at the beginning of this section, the above FIT values of the 4 fault mechanisms are added together to get the total ROB FIT ($FIT_{Total\text{-}ROB}$). Then we can compute the probability of each fault category ($P_{Transient}$, $P_{Intermittent}$, and $P_{Permanent}$). Now the $FIT_{Total\text{-}ROB}$ value specifies the rate at which faults are injected in the ROB, and the 3 probabilities are used to determine the probability of occurrence of each fault type. While $FIT_{Total\text{-}ROB}$ is the most accurate way to determine when to inject a fault, these values are usually small and occur exceedingly rarely compared to processor clock period. The only escape is to accelerate our simulations, so we select to inject a fault in the ROB using a uniform random variable with a mean of 10 million cycles. In other words, in every cycle during our simulation the probability of a fault being injected in the ROB is 10^{-7}. We also selected the ROB entry which will be allocated next to inject the fault. By this approach the ROB entry number selected is also randomly selected. Once we decide to inject a fault we use the relative probability of occurrence of the three fault types (computed as described earlier) to decide *which* fault type to inject.

The last aspect in fault injection is to decide how long a fault will persist (i.e. fault period). The fault period depends on fault type. If the fault injected is transient then it will disappear after the RAE handler completes its execution and enables the faulty entry. If the fault injected is permanent, it will persist for the entire simulation. Injecting intermittent faults (i.e. NBTI faults) require computing the length of time the fault persists. In our simulation we use bit flipping frequency, number of times a bit has flipped from 1 to 0 or vice-versa, inside a ROB entry to determine the fault period. If bits flip quite often then NBTI severity is low else it grows. If flipping frequency is greater than 75% of total simulation cycles since last fault in the same ROB entry, level 1 intermittent fault is injected with a fault period of 1 million cycles. If flipping frequency is between 50-75% of total simulation cycles since last fault in the same ROB entry, level 2 intermittent fault is injected. Level 2 fault period is 2 million cycles. Level 3 intermittent fault is injected when flipping frequency is between 25-50% of total simulation cycles since last fault in the same ROB entry. The fault period is 4 million cycles. In case flipping frequency is extremely low (less than 25% of total simulation cycles since last fault in the same ROB entry), level 4 intermittent fault is injected. Level 4 fault is treated as permanent. In essence in our simulation experiments level 1, level 2, and level 3 intermittent faults injected in ROB will disappear after the faulty ROB entry is de-configured for at least 1, 2, and 4 million cycles, respectively.

C. Simulation Results

For our simulation experiments, we chose 8 SPEC CPU2000 benchmarks, 4 Integer benchmarks: Bzip2, Crafty, Gzip, and Parser and 4 floating point benchmarks: Apsi, Mgrid, Swim, and Wupwise. We used single SimPoint to determine the simulation point to start detailed simulation for 3 billion committed instructions. We present our results in 2 subsections: First, we compare the reliability of base and RAE machines using the FIT values for TDDB, NBTI, and EM fault mechanisms. Second, we discuss the performance impact of the RAE mechanism.

In the base machine, there is no handling mechanism available for RAEs. Hence, whenever a fault is detected in one of the ROB entries in the base machine, we flush the processor and start executing from the faulty instruction by

allocating it to the ROB entry next to the faulty one to ensure forward progress. This is done for all types of faults. As one would expect, on an intermittent fault the machine encounters multiple pipeline flushes, once every time the entry is used. On the other hand, the RAE machine uses the RAE handling mechanism to use the appropriate solution based on fault category.

1) Reliability Evaluation: In this subsection, we compare the reliability of the base and RAE machines using the FIT values of TDDB, NBTI, and EM fault mechanisms. Fig. 2 shows the FIT values for different benchmarks at different time instances. There are 8 graphs (one for each benchmark). In all graphs, the X-axis is the simulation time and the units are 10^{10} cycles (10 Billion cycles). The Y-axis is FIT values. In each graph there are 6 curves that are labelled with two characters separated with an underscore. The first character indicates the machine type: "R" stands for RAE machine and "B" for base machine. The second character represents the fault mechanism: "T" stands for TDDB, "N" for NBTI, and "E" for EM. For example "R_T" is the RAE machine's TDDB FIT. Each curve runs for different simulation cycle lengths. In the base machine when all ROB entries have either an intermittent or permanent fault the simulation cannot proceed any further. Hence, the machine simply crashes at that time. This scenario can also occur in the RAE machine when the number of disabled/de-configured entries equal to ROB size. However, the likelihood of this event is small. RAE machine recovers faster from intermittent faults since turning OFF a ROB entry increases the chance of its recovery. Hence, in our simulations the base machine fails to complete the execution of 4 benchmarks for 3 Billion committed instructions. On the other hand the RAE machine completed the simulation in all cases, except for Gzip where it failed after executing 2.8 Billion instructions after it ran out of ROB entries. The exact numbers of instructions committed per each benchmark are shown in fig. 3. What is interesting to note from the graphs is that base machine executed for much longer simulation cycles even when fewer number of instructions is committed. Not surprisingly, the base machine suffers from severe pipeline flush penalties as it uses pipeline flush as the only corrective action to deal with faults. Hence, even though base machine committed fewer instructions before a crash it takes far more cycles to execute them. The RAE machine on the other hand commits the specified 3 Billion instructions and it executes them much faster than the base machine.

a) TDDB: The primary observation we note is that the $FIT_{TDDB\text{-}ROB}$ values are less than $FIT_{NBTI\text{-}ROB}$ and $FIT_{EM\text{-}ROB}$ values especially on the base machine. Referring to FIT models which were discussed earlier, TDDB FIT depends on the ROB activity *per cycle* which stays almost stable throughout the simulations for all benchmarks on both machines. NBTI and EM FIT models depend on the cumulative ROB activity which always grows with time. Hence, TDDB FIT does not change much with time in our simulations.

b) NBTI: Fig. 2 shows that for all benchmarks except for Gzip, the $FIT_{NBTI\text{-}ROB}$ on base machine increases with time as the average ROB temperature increases then stabilizes to some value as the average ROB temperature reaches a steady state value. The $FIT_{NBTI\text{-}ROB}$ is mostly affected by the change in average ROB temperature due to the exponential relationship between parameter K in (3) and the average ROB temperature. So, when comparing RAE and base machine we see that the RAE machine $FIT_{NBTI\text{-}ROB}$ is less than the base machine for all benchmarks except Gzip. For the 7 benchmarks, the total ROB activity on base machine is always higher than RAE machine, as shown with ROB activity values of Apsi benchmark in fig. 4. Each curve in fig. 4 is labelled by the machine type (RAE or Base) followed by a character: "T" for true ROB activity, "M" for mispredicted ROB activity, and "W" for wasted ROB activity. Since the base machine has no corrective actions for NBTI faults, the number of ROB flushes due to reliability faults (i.e. wasted ROB activity) will be much higher compared to RAE machine. Higher ROB activity leads to higher average ROB temperature and hence makes the ROB more susceptible to NBTI faults. On the other hand, RAE machine deploys FDU de-configuration mechanism to reverse the effects of NBTI faults which reduce the wasted ROB activity and hence make ROB more resilient to NBTI faults. The Gzip behaviour is explained exactly the same as in TDDB discussion.

Gzip exhibits vastly different behaviour in the initial phase of simulation, up until 2 Billion simulation cycles. During the initial phases of Gzip simulation RAE machine experiences higher true ROB activity as it deals with fewer pipeline flushes compared to base machine. As we will show in the next section, the IPC on RAE machine is much higher during the early phase of simulation than the base machine. Higher IPC results in higher true ROB activity. Even though base machine suffers more flushes the increase in wasted ROB activity on base machine is not enough to offset the increase in the true ROB activity on RAE machine. As a result the total ROB activity on RAE machine becomes higher than base machine for about 2 billion simulation cycles, which causes higher average ROB temperature and consequently higher $FIT_{TDDB\text{-}ROB}$ values. As the simulation of the Gzip progresses, the true ROB activity on RAE machine starts to decrease as the number of de-configured ROB entries increases. At that point, the increase in the wasted ROB activity on base machine offsets the increase in true ROB activity on RAE machine and causes the total ROB activity, average ROB temperature, and $FIT_{TDDB\text{-}ROB}$ values of base machine to higher. The ROB activity numbers for Gzip benchmark are given in fig. 4.

c) EM: Two main points can be observed from EM curves in fig. 2. The first point is that for all benchmarks on both RAE and base machines, the $FIT_{EM\text{-}ROB}$ increases with time. For Bzip, Crafty, Gzip, Parser, Swim, and Wupwise benchmarks the $FIT_{EM\text{-}ROB}$ increases from 0 to almost 0.35 on base machine. For Apsi and Mgrid benchmarks, the $FIT_{EM\text{-}ROB}$ increases from 0 to almost 0.85 on base machine.

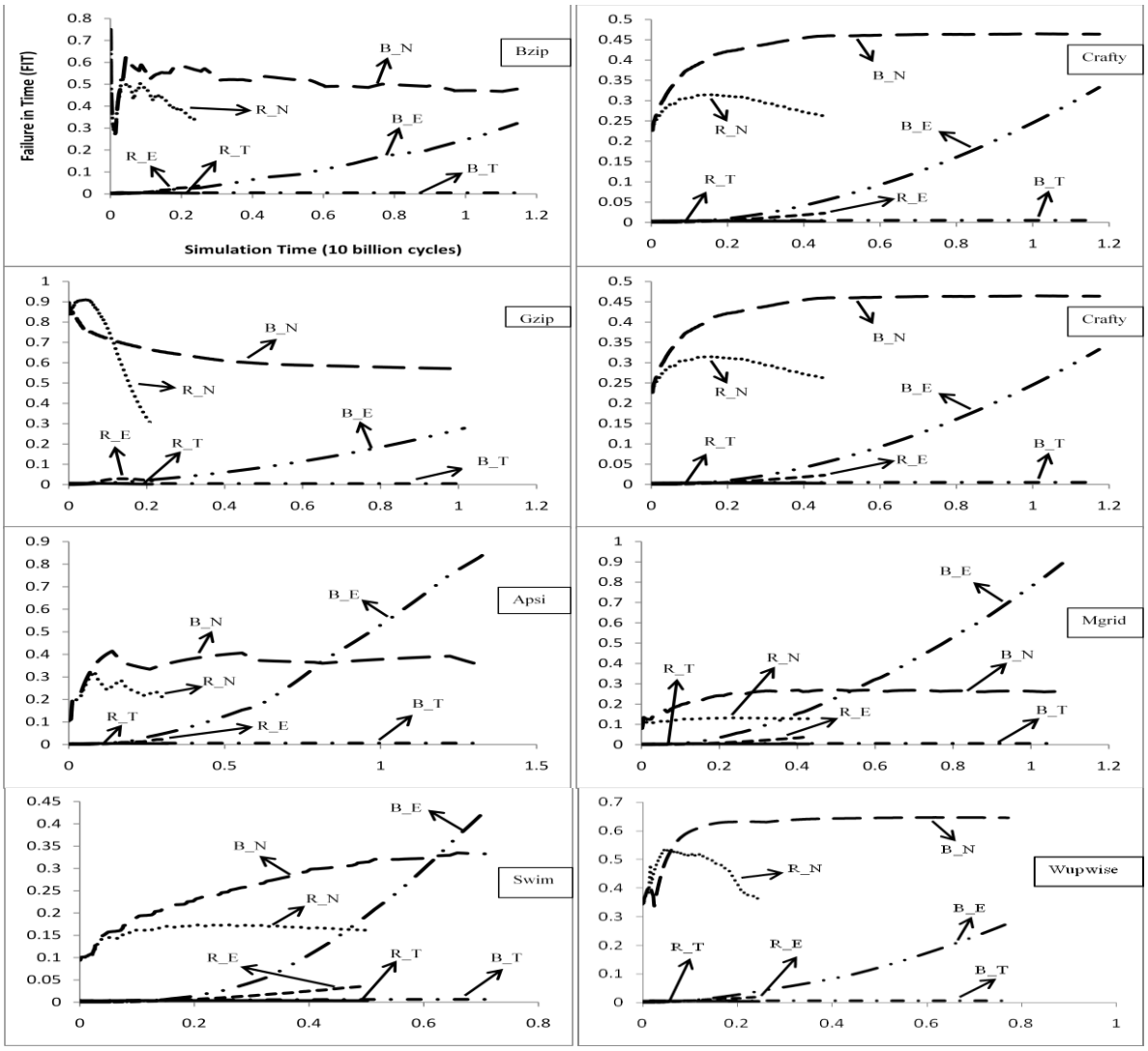

Figure 2. FIT values of TDDB, NBTI, and EM

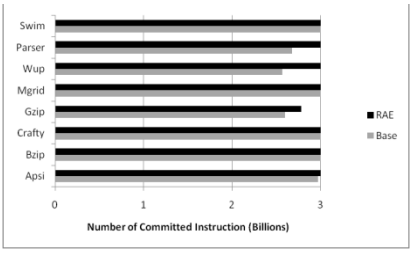

Figure 3. Total Number of Committed Instruction

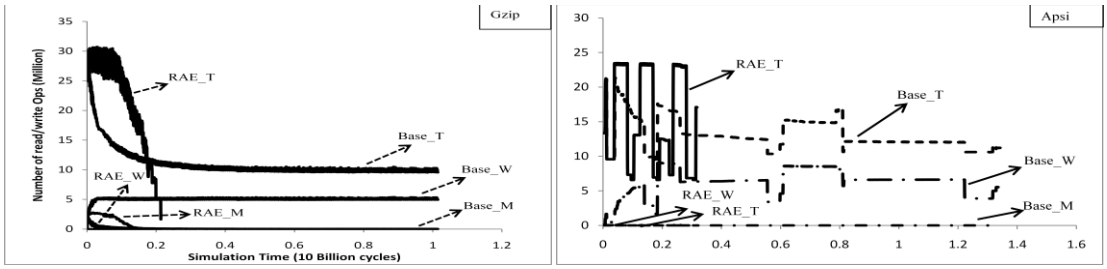

Figure 4. ROB Activity

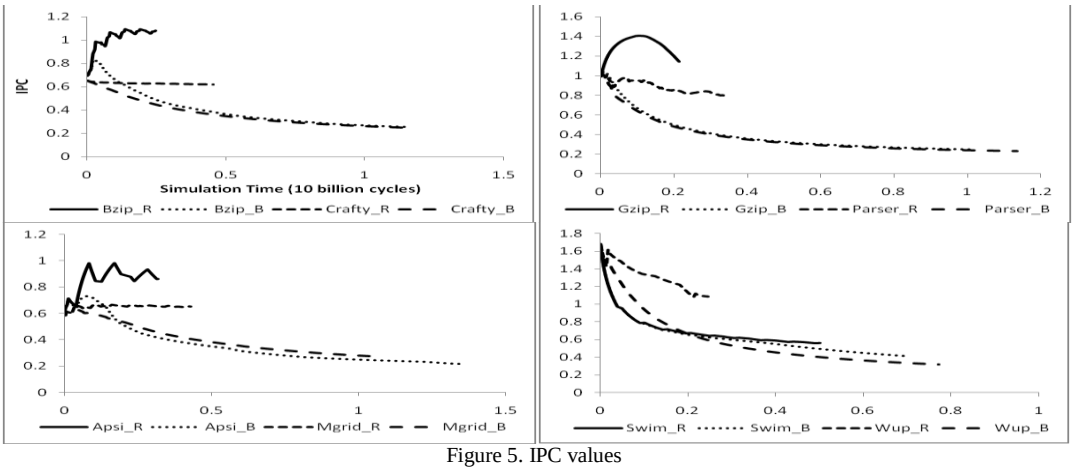

Figure 5. IPC values

From (6), $FIT_{EM\text{-}ROB}$ has a quadratic relationship with the cumulative number of reads and writes operations in the ROB (from the beginning of the simulation and up to the specific point where $FIT_{EM\text{-}ROB}$ is computed, i.e. total ROB activity). For all benchmarks on both machines the total ROB activity increases with time; hence $FIT_{EM\text{-}ROB}$ simply increases with the amount of time the machine is operational. As execution time on base machine is greater than execution time on RAE machine for all benchmarks, we expect the $FIT_{EM\text{-}ROB}$ to reach much higher values in base machine.

The second point is that for all benchmarks except Gzip, the $FIT_{EM\text{-}ROB}$ values on base machine are greater than $FIT_{EM\text{-}ROB}$ values on RAE machine. This point is the same as the second observation in NBTI discussion. As we introduce more faults in the ROB, the base machine starts to suffer from higher wasted ROB activity due to the lack of any corrective action. The total ROB activity becomes higher on base machine compared to RAE machine which also causes the average ROB temperature to increase as well. The compound result of higher ROB activity and higher average ROB temperature on base machine results in higher $FIT_{EM\text{-}ROB}$ values. The different behaviour of Gzip can be traced back to the same discussion that we provided in analyzing NBTI results.

2) Performance Evaluation: We initially thought that RAE mechanism will cause some performance degradation due to the de-configuration process which reduces the number of active ROB entries; hence, less in-flight instruction and less IPC. However, the performance results show that RAE machine has better performance than base machine. The execution time reduction factor ranges from 1.5 for Swim benchmark to 5 for Bzip benchmark. As we inject more faults in the ROB of both the base and RAE machine, the temporal and permanent de-configuration of ROB entries in RAE machine causes the IPC to drop but at a rate much less rate than the rate at which IPC of base machine drops due to the increase in wasted ROB activity. In other words, having fewer active ROB entries in RAE machine does not necessarily mean lower IPC. In fact, without temporal de-configuration most of the work done on base machine is wasted as more ROB entries are faulty.

Fig. 5 shows the IPC values of both base and RAE machines for all benchmarks throughout the simulation measured at the granularity of 5 million cycles. Fig. 5 has four graphs with two benchmarks in each graph. Each benchmark is appended with *_B* to represent IPC on the base machine. We use *_R* to represent IPC on the RAE machine. The X-axis is the simulation time in 10 Billion cycles and the Y-axis is IPC values. For all benchmarks, the IPC values of RAE machine are greater than the IPC values of base machine, nearly for the entire simulation duration. IPC values of base machine for all benchmarks decrease with simulation time which is expected due to the increase in the number of faulty ROB entries and the lack of any corrective actions. The effectiveness of the RAE approach can be best seen in Bzip and Apsi benchmarks. For these benchmarks the IPC values of RAE machine increase with time during the early stages of the simulation then start to toggle between increase and decrease to the end of the simulation. The toggling of IPC in these benchmarks during simulation is the result of RAE actions. When ROB entries are temporarily de-configured in response to an intermittent fault the IPC drops. But as these entries are reconfigured after recovery phase completes IPC increases. For Gzip the base machine IPC suffers right from the start. Further analysis showed that this benchmark has plenty of instruction level parallelism at the beginning of the selected simulation point. The base machine's repeated pipeline flush however caused the net IPC to drop. In the RAE machine the instruction level parallelism is well exploited during the beginning of simulation. But as the RAE machine continued to operate more ROB entries were either temporarily or permanently de-configured. Hence, IPC was impacted by the shrinking ROB size.

In conclusion, we showed that using RAE mechanism help increase the reliability of the ROB against TDDB, NBTI, and EM fault mechanisms. The permanent de-configuration of ROB entries that exercise permanent fault makes the ROB more resilient to permanent faults by isolating the faulty entry to avoid excessive flushes. At the same time we achieve a speed up that ranges between 1.5

and 5 for different benchmarks. Although in our experimental design we only applied RAE to the ROB, we expect to get the same benefits when applying RAE to other circular buffer array structures (LSQ, IFQ), tabular array structures (RS), and non storage structures.

IV. RELATED WORK

There has been considerable research that focuses on tolerating transient and permanent faults. SWAT uses anomalous software behavior to indicate the presence of hardware faults [10]. There are many advantages for this high level detection mechanism: Low detection overhead and ignoring masked faults. However, the latency from the time the architectural state is corrupted to the time the fault is detected is in the range of microseconds for GHz processors; so the checkpointed state information must be kept long after the instruction commits which increases the memory overhead of the checkpointing mechanism. RAE share the same advantages as SWAT through using low cost detection mechanisms which require minimum hardware such as parity for storage structure and partial redundant execution using look-ahead circuits to detect faults in data and control intensive logical blocks (ALUs, Instruction schedulers,…etc). In addition, RAE minimizes the memory overhead associated with rolling back the execution by handling reliability faults when instructions commit. SWAT can only detect transient and permanent faults by re-executing the faulty portion of the program on the expected faulty hardware and a fault-free hardware and compare the two results. Solely depending on the re-execution mechanism described above is too aggressive as many intermittent faults which persist for long periods will be interpreted as permanent faults. RAE uses history based classification to differentiate between transient, intermittent, and permanent faults. Hence, RAE gives a range of different actions from simply leverage the existing roll back mechanisms in case of transient faults to full isolation of the faulty FDU in case of permanent faults. In case of intermittent faults, RAE allows us to de-configure the faulty FDU temporarily. Turning OFF faulty FDUs may help reverse the bad effects of some intermittent faults such as NBTI faults. For intermittent faults such as EM and TDDB faults, turning OFF faulty FDUs may help improve their life time by avoiding continuous stress conditions.

In [3], Bower et al proposed Self-Repairing Array Structures (SRAS): a light weight hardware based mechanism to tolerate hard errors in microprocessor array structures. SRAS is implemented in conjunction with the DIVA core [1]. DIVA core is a special checker processor attached at the commit stage of a more aggressive microprocessor. DIVA is responsible for detecting and correcting computation and communication faults by re-executing every retired instruction and instantiating the recovery action (flushing the aggressive main microprocessor) every time a fault is exercised. SRAS is used to map out entries of array structures that exercise a hard fault, hence removing continuous flushing due to such faults. SRAS uses a handful of check rows to detect faults in each array structure. Every write operation to a row in a certain array structure is duplicated to a check row, and then data is read from both rows and compared –off the critical path- to detect errors. As the number of faults detected from a specific row reaches a pre-defined threshold value, the row is believed to have a permanent fault and is permanently mapped out. Otherwise, the fault is assumed to be transient. This detection mechanism is expensive in terms of both area and power consumption. On the other hand, RAE uses cost effective fault detection mechanism for array structures represented by the use of a single parity bit for every single row which saves a lot of area compared to SRAS. In addition, RAE does not require a duplicated write or a comparison operation, hence saving power consumption. Another major limitation of using check rows is that they may become faulty themselves; as all check rows exercise hard faults, the detection mechanism goes down.

Bower et al proposed an online diagnosis mechanism of hard faults in microprocessors [4]. They implement a mechanism to diagnose hard faults at the field de-configurable unit (FDU) granularity. As before, Bower et al depends on DIVA checker to provide error detection and correction. RAE depends on previously known, simple and reliable detection mechanisms applied for each FDU. The online diagnoses mechanism uses small hardware saturated counters for every FDU. During instruction execution, the instruction occupancy of the core structures is tracked using a stream of bits. When the DIVA checker detects a fault in an instruction, all counters of the core structures that the instruction used are incremented by 1. When a specific FDU counter reaches a pre-defined threshold value, a permanent fault is detected in that FDU and it is permanently mapped out. Otherwise, the fault is assumed to be transient. One major limitation of the online diagnosis mechanism is the assumption of independent resources usage. Without this assumption, we could end up mapping out fault-free FDU if two resources happen to be used together most of the time. Despite the modifications in the resource scheduling algorithms that Bower et al suggested, it remains impossible to prevent such false mapping out. RAE does not suffer from this drawback, because faults are detected at the output of each FDU that the instruction uses.

RAE uses the same mapping out mechanism for circular buffer arrays proposed by SRAS [3]. Also RAE uses the same mapping out mechanism for tabular array structures and non storage structures proposed by the online diagnosis mechanism [4]. In addition to the differences discussed above between SRAS and RAE, and online diagnosis mechanism and RAE, There is one more thing worth mentioning, both SRAS and the online diagnosis mechanism classify the faults as either permanent or transient faults based on counter values. On the other hand, RAE classifies the fault as transient, intermittent or permanent based on the history of the faulty FDU. This may cause some intermittent faults in SRAS and online diagnosis mechanism to be falsely classified as permanent faults and hence result in the faulty FDU being mapped out permanently.

V. CONCLUSION

In this paper we propose reliability aware exceptions (RAE), a new class of exceptions that run entirely in software and require minimal hardware cost. RAE has the ability to classify the fault into one of the 3 categories (transient, intermittent, and permanent) and then choose the optimal corrective action which helps reverse or slow down the impact of the fault. RAE fault classification algorithm uses fault history logs which maintain records of previous detected faults indexed by a unique FDU identification number (FDU_ID). Corrective actions range from simple flushing and re-execution in case of transient faults to permanent de-configuration of the faulty FDU in case of permanent faults.

In addition, we present a new fault injection approach to mimic the physical phenomena of the three fault mechanisms: TDDB, NBTI and EM. We measure the environmental and stress conditions continuously during simulation. These measurements are used to compute FIT_{TDDB}, FIT_{NBTI}, and FIT_{EM} values which can then be used to get the probability distribution of each fault mechanism.

In this paper we focused on RAE implementation to protect the ROB structure. However, we have shown that the implementation can be easily extended to protect other storage and non storage structures incurring negligible area overhead compared to reliability and performance improvements.

REFERENCES

[1] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In Proc. of the 32nd International Symposium on Microarchitecture, pages 196–207, Nov. 1999.

[2] J. R. Black. Electromigration-A Brief Survey and Some Recent Results. Electron Devices, pages 338 – 347, Apr 1969.

[3] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating Hard Faults in Microprocessor Array Structures. In Proceedings of International Conference on Dependable Systems and Networks, p.51, June 28-July 01, 2004.

[4] F. A. Bower, D. J. Sorin, and S. Ozev. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. In Proceedings of the 38th International Symposium on Microarchitecture, pages 197-208, Nov 2005.

[5] G. J. Briggs, E. J. Tan, N. A. Nelson and D. H. Albonesi. QUILT: A GUI-based Integrated Circuit Floorplanning Environment for Computer Architecture Research and Education. In Workshop on Computer Architecture Education, pp. 26-31, Jun 2005.

[6] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In Proceedings of the 27th Annual international Symposium on Computer Architecture, pages 83-94, 2000.

[7] D. Burger and T. M. Austin. The SimpleScalar tool set, Computer Architecture News, v.25 n.3, p.13-25, June 1997.

[8] M. Choudhury and K. Mohanram. Timing-driven optimization using lookahead logic circuits. In Proceedings of 46th Design Automation Conference, Sept 2009.

[9] W. Huang, K. Sankaranarayanan, K. Skadron, R. J. Ribando, and M. R. Stan. Accurate, Pre-RTL Temperature-Aware Design Using a Parameterized, Geometric Thermal Model. IEEE Trans. Comput. 57, 9 (Sep. 2008), 1277-1288.

[10] M. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve S. V., and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. Computer Architecture News, v.36, n.1, 265-276. March 2008

[11] U. Raju, A. Kaisare, D. Agonafer, A. Haji-sheikh, G. Chry sler, R. Mahajan. Multi-Objective optimization entailing computer architecture and thermal design for non-uniformly powered microprocessors. Thermal and Thermomechanical Phenomena in Electronic Systems, pages 432 – 440, 2008.

[12] K. P. Rodbell, A. J. Castellano and R. I. Kaufman. AC electromigration (10MHz–1GHz) in Al metallization. Proceedings of the fourth workshop on stress induced phenomena in metallization, pp. 212-223, January 1998.

[13] R. Rodriguez et al. The impact of gate-oxide breakdown on SRAM stability. IEEE Electron Device Letters, Vol. 23, No. 9, pp. 559-561, September 2000.

[14] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer. An Experimental Study of Soft Errors in Microprocessors. IEEE Micro, v.25 n.6, p.30-39, 2005.

[15] S. K. Sastry Hari, M. Li, P. Ramachandran, B. Choi, and S.V. Adve. mSWAT: low-cost hardware fault detection and diagnosis for multicore systems. In Proceedings of the 42nd International Symposium on Microarchitecture, pages 122-132, 2009.

[16] D. K. Schroder and J. A. Babcock. Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing. Journal of Applied Physics, Vol. 94, No. 1, July 2003.

[17] International Technology Roadmap for Semiconductors, 2007. http://www.itrs.net.

[18] K. Shi, D. Howard. Sleep Transistor Design and Implementation Simple Concepts yet Challenges to Be Optimum. VLSI Design, Automation and Test, 2006 International Symposium, pages 1–4.

[19] J. Shin, V. Zyuban, Z. Hu, J. A. Rivers, and P. Bose. A Framework for Architecture-Level Lifetime Reliability Modeling. pp.534-543, 37th Intl. Conference on Dependable Systems and Networks, 2007.

[20] 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, pages 73–82, Oct 2006.

[21] J. Srinivasan, S. V. Adve, P. Bose and J. A. Rivers. The case for lifetime reliability-aware microprocessors. In Proceedings of International Symposium on Computer Architecture, pp. 276-287, June 2004.

[22] R. Vattikonda, W. Wang, and Y. Cao. Modeling and Minimization of PMOS NBTI effect for Robust Nanometer Design. In Proceedings of the 43rd annual conference on Design automation, pages 1047–1052, July 2006.