Intermittent Hardware Errors Recovery: Modeling and Evaluation

Layali Rashid, Karthik Pattabiraman and Sathish Gopalakrishnan
The University of British Columbia, Canada
{lrashid, karthikp, sathish}@ece.ubc.ca

***Abstract*—The frequency of hardware errors is increasing due to shrinking feature sizes, higher levels of integration, and increasing design complexity. Intermittent errors are those that occur non-deterministically at the same location. It has been shown that intermittent hardware errors contribute to about 39% of the total hardware failures. Intermittent faults have characteristics that are different than transient and permanent errors, which makes it challenging to devise efficient recovery techniques for them.**

In this paper, we evaluate the impact of different intermittent error recovery scenarios on the processor performance. To achieve this, we model a system that consists of a fault-tolerant multicore processor subject to intermittent faults. Our fault models are based on insights from related work at the physical level. We find that the frequency of the intermittent error and the relative importance of the error location play an important role in choosing the recovery action that maximizes the processor's performance.

***Index Terms*—Intermittent hardware faults, recovery, stochastic activity network, fault model, transistor wearout.**

I. Introduction

Intermittent hardware errors occur sporadically at the same hardware location, and persist for one or more (but finite number of) clock cycles. Recent work has shown that intermittent hardware failures are prevalent in commodity processors [1], [2]. Such faults result from combinations of extremely small transistor dimensions, process variations and abnormal operating conditions such as high temperature and/or high voltage, and they are likely to increase in frequency in future processors [3], [4].

In this work, we focus on intermittent error recovery, i.e., the action that should be taken after detecting an intermittent error to remedy the effects of the error and prevent its occurrence in the future. Since repairing (or fixing) the defective circuit in commodity processors is prohibitively expensive, recovery from hardware errors primarily consists of error mitigation.

Recovery from transient errors consists of rolling back to the last checkpoint and re-executing the program because the error is unlikely to recur. On the other hand, recovery from permanent errors consists of disabling the defective part of the processor since the error persists in that part forever. Intermittent errors exhibit characteristics between these two extremes (appear non-deterministically at the same location). However, recovery from intermittent errors cannot simply be similar to transient errors (for low intermittent fault rates) or permanent errors (for high intermittent fault rates), as we show later in this paper.

The research question we address is: *What is the recovery action that maximizes the performance of an intermittent-error prone processor?* Towards answering this question, we make the following contributions:

(1) Build a model of a chip multiprocessor running a parallel application. The model is built using Stochastic Activity Networks [5], and includes error detection, discrimination (from other types of hardware errors), diagnosis and recovery. We model the entire system because we find that configuration parameters in data checkpointing, for example, can affect the choice of the recovery option.

(2) Propose intermittent fault models that abstract real intermittent faults at the system level. Intermittent faults are fundamentally different from permanent and transient faults, and therefore cannot be accurately modeled using permanent or transient fault models. Further, due to the lack of information about intermittent faults' exact characteristics, it is challenging to model such faults at a high level. Based on insights from related work [6]–[8], we build multiple fault models that abstract physical fault models and at the same time we prune down the space of system configurations to a manageable set of parameters that we can simulate.

(3) Simulate the system (which consists of the two models described above) to evaluate the performance of a processor after applying different recovery options that include permanent/temporary shutdown of cores/microarchitectural units of a processor. Moreover, we study the sensitivity of our results to the models' parameters.

Our salient findings are as follows:

- The frequency of an intermittent error and the relative importance of the defective part in the processor play an important role in finding the recovery action that maximizes the processor's performance. Therefore, intermittent faults cannot simply be treated as transient faults or permanent faults without regard to their parameters.
- Unlike permanent errors, we find that if a core is exhibiting high intermittent failure rates, then the best recovery action depends on the relative importance of the defective part of the processor. Further, we find that low intermittent failure rates can be tolerated by a program rollback to the last checkpoint.
- Contrary to what other researchers have suggested [9], we find that a permanent shutdown of the intermittent error-prone part of the processor results in a slight improvement of the processor's performance compared to

the temporary shutdown of the same part.

II. Intermittent Faults

A. Definition

We define an intermittent fault as one that appears sporadically at the same hardware location, and lasts for one or more (but finite number of) clock cycles. This is consistent with definitions in prior work [2], [9]. The main characteristic of intermittent faults that distinguishes them from transient faults is that they occur repeatedly at the same location, and are caused by an underlying hardware defect rather than a one-time event such as a particle strike. However, unlike permanent faults, intermittent faults appear non-deterministically, and only under certain operating conditions.

B. Causes

The major cause of intermittent faults is device wearout, or the tendency of solid-state devices to degrade with time and stress. Wearout can be accelerated by aggressive transistor scaling which makes processors more susceptible to extreme operating condition such as voltage and temperature fluctuations [3], [4]. Many well studied mechanisms such as gate-oxide breakdown, negative bias temperature instability (NBTI), hot-carrier injection (HCI) and electromigration [4] occur due to wearout.

Studies have shown that errors generated by NBTI are intermittent because up to 40% of the NBTI damages are reversible after stress removal (typically, high temperatures). The remaining 60% of the damages are permanent (similar behavior has been observed for HCI) [10]. Time-dependent dielectric breakdown (TDDB) results in intermittent errors because it appears as Soft Dielectric Breakdown (SDB) at nominal operating conditions, which is less damaging, then progresses to Hard Dielectric Breakdown (HDB) if the stress conditions persist [6], [8]. In the long term, such faults may eventually lead to permanent defects.

Another cause of intermittent faults is manufacturing defects that escape VLSI testing [2]. Often, deterministic defects are flushed out during such testing and the ones that escape are non-deterministic defects, which emerge as intermittent faults.

Finally, design defects can also lead to intermittent faults, especially if the defect is triggered under rare scenarios or conditions [11].

C. Statistics of fault occurrence

Few field studies have been conducted on intermittent error rates. Constantinescu [2] found that 6.2% of the hardware errors in memory subsystems are intermittent. However, he does not present any data for processor faults. A recent study by Nightingale et al. [1] analyzed error logs sent by the Microsoft Windows Error Reporting (WER) program from 950,000 personal computers. They found that, of the hardware errors reported about microprocessors, approximately 39% are intermittent. However, they only consider a small class of processor errors, and only those that cause operating-system crashes. As a result, they under-estimate the number of intermittent errors that occur in the field.

Fault distribution: Studies that focus on wearout failures found that the distribution of aging failures follow either log-normal [12]–[14] or Weibull distributions [15]. These studies focus on the phase in which the aging errors persist for long periods and eventually lead to permanent errors. In contrast, the focus of our work is on intermittent faults that have not progressed to permanent faults yet.

D. Fault models

Due to the lack of intermittent faults' field studies in commodity processors, there are no definite answers as to when exactly they occur, how frequently they occur and the length of their occurrence. In this section, we rely on prior work to build approximate intermittent fault models. Moreover, we study the sensitivity of our results to fault models and failure rates (Section V-A).

We make the following assumptions in our fault models:

- At any time, a microarchitectural unit may be affected by at most a single intermittent fault.
- At any time, at most a single microarchitectural unit may be affected by an intermittent fault.

The intermittent faults considered in this paper are caused by temperature/voltage fluctuations and wear out. We do not consider intermittent errors caused by manufacturing defects or design defects due to lack of information about their distribution. Moreover, since the focus of this paper is on processor faults, we do not consider errors that occur in the memory and I/O hierarchy. Such errors are usually tolerated by Error Correcting Codes and Cyclic Redundancy Checks.

We consider three fault models to characterize intermittent faults as follows:

- **Base fault model:** In this model, we assume that each core experiences faults that are distributed exponentially with an MTTF of 6.56 years. This MTTF is found by Nightingale et al. [1] using a large-scale study on data obtained from the Windows Error Reporting System. Therefore, intermittent faults in this model occur rarely. The machine in this model is hardened against temperature and voltage fluctuations, infant mortality and wearing out. Such processors can be fabricated using large transistors, for example.
- **Exponential fault model:** Similar to the base fault model, each core starts its lifecycle with a base MTTF of 6.56 years. However, at some point during its life time the core experiences high temperature and voltage fluctuations, but these fluctuations do not permanently affect transistor performance. In other words, the transistors in this model are sensitive to temperature/voltage swings but there is no wearing out. Fault arrivals are modeled as a Poisson process. The duration of a fault burst is also assumed to follow an exponential distribution.
- **Weibull fault model:** This model is similar to the exponential fault model in that each core starts its lifecycle

with a base MTTF of 6.56 years and during its life time it experiences extreme temperature/voltage instabilities. However, in this model such hikes in temperatures and voltages will lead to transistor wearouts [6]–[8], [10]. Hence, we model the occurrence of fault bursts using Weibull distribution with a shape parameter of 1.4 [16] and a variable scale parameter depending on the MTTF. Similar to the previous model, during each error burst, the fault is modeled using exponential distribution.

III. System Description

Our target system is an error-tolerant chip multiprocessor, where two to thirty-two cores reside on the same chip. A study of the efficiency of a recovery technique of such a system should not only include a model of the recovery technique, but also a model of all other fault-tolerance techniques at different levels of granularity. This is because these tolerance techniques include design parameters that may affect the choice of the appropriate recovery action. Our model consists of fault detection, discrimination, diagnosis and recovery.

We start this section by listing our definitions and assumptions (Section III-A). Then we present an overview of the fault-tolerance techniques that are common to all recovery scenarios (Section III-B). After that we detail the different recovery models that we consider: (1) rollback-only recovery: where no reconfiguration action is conducted. A simple rollback to the last checkpoint is applied upon each error detection. (2) Core-level reconfiguration: where the defective core is disabled upon error detection and discrimination. (3) Unit-level reconfiguration: where a fine-grained reconfiguration at the level of the defective microarchitectural unit is conducted upon error detection, discrimination and diagnosis.

Finally, we describe our Stochastic Activity Network (SAN) models in Section III-D (we use SANs to build the model of the system).

A. *Definitions and assumptions*

We follow the standard definitions of the fault tolerance terms, a summary of the terms we use is available in Table I.

TABLE I: Terms and definitions.

Term	**Definition**
Detect	to find that an error affected a program such that the program's state has been changed erroneously.
Diagnose	to isolate the physical location of an error, a core or a microarchitectural component in this work.
Discriminate	to distinguish between transient, intermittent and permanent hardware error.
Recover	to remedy the effects of an error on program state by restoring the program's state to the last saved checkpoint and possibly disabling the defective physical part of the processor.

To evaluate the performance of a processor under different recovery scenarios, we use the term *useful work* [17]. Useful work is the fraction of effective work a processor accomplishes in a certain time duration. It does not include work repeated because of a failure that occurs before saving work to a checkpoint, nor does it include the work done while saving checkpoints, recovering from errors and diagnosing and reconfiguring the processor. More details about how we measure the useful work are available in Section IV.

We make the following assumptions in the model:

- Errors covered are due to intermittent hardware faults only. However, we model a fault discrimination technique to distinguish such failures.
- Intermittent failures can occur during checkpoint recording and during processor reconfiguration.
- Intermittent failures cannot occur during program rollback to the last checkpoint. They also cannot occur during error diagnosis process. This assumption is reasonable because most diagnosis techniques can be performed using other healthy cores or specialized hardware [12], [18].

A discussion of the impact of these assumptions on the applicability of our model to real processors is available in Section VI.

B. *Overview of an error-tolerant core*

We use state transition diagrams to describe system evolution (Figure 1 describes the lifecycle of a core). Note that this diagram is shown for exposition only, and the actual model is built using SANs. This part of the model is common to all three recovery scenarios.

In this system, a multithreaded program runs in an on-chip-multiprocessor with two to thirty-two cores (Box 1). A coordinated checkpoint is recorded periodically (Box 2). All cores use coordinated checkpointing, i.e., a checkpoint is taken for all cores at the same time [17]. The details of the coordinated checkpointing implementation are out of this paper's scope. Coordinated checkpointing implies that all cores on a chip are running one parallel program. If a core fails, the program will be restored to the previous checkpoint (hence, all cores perform recovery) and any work that has not been checkpointed is lost. *This model is a representative of modern workloads where massively parallel programs are running across all cores simultaneously.* While the system is running (either executing a program or storing a checkpoint), an intermittent fault can occur in any unit of the processor in any of the cores.

If the intermittent fault is activated, i.e., if this fault manifests itself to the program, it can cause the following outcomes [19] [20]: (1) benign, which means that the program affected by this error will continue running and generating correct results, (2) silent-data corruption, which means that the program will continue running but its state is corrupted by the error. At a later stage this error might be detected by software/hardware detector (Box 3), or (3) the fault activates a hardware trap and the program crashes (e.g., program accesses memory using invalid address) (Box 3). In our model, if an error is activated, is not a benign error and is detected then this behavior is represented by a transition from Box 1 or 2 to Box 3, otherwise, the program can continue running as

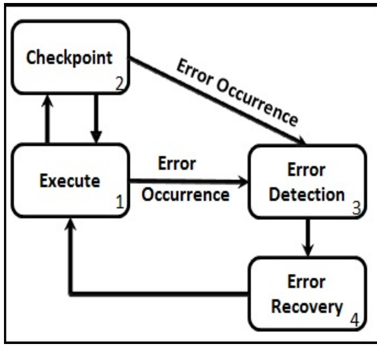

Fig. 1: An overview of an error-tolerant core with rollback-only recovery.

usual. When an error is detected or a program crashes then the program rolls back to the last checkpoint to recover from the effects of the error (Box 4).

C. Recovery scenarios

We now describe different scenarios for intermittent-error recovery. As mentioned in the previous section, all the models below share the same mechanisms for error detection and checkpointing.

Rollback-only recovery: In this basic model (Figure 1), no recovery technique is applied other than rolling the program to a checkpoint. Therefore, no error discrimination technique is required. Upon error detection, recovery using the last stored checkpoint is conducted (Box 4) and then the system continues running as usual (Box 1).

Core-level reconfiguration: In this model (Figure 2), upon error detection a fault-discrimination technique is applied. We use the Alpha count-and-threshold error discrimination mechanism proposed by Bondavalli et al. [16] for fault discrimination (Box 5). The recovery action for this particular scenario is to disable the core (Box 6). A diagnosis technique is not required in this model since each core has its own error-distinguishing technique which identifies the intermittent-error prone core. The duration of the core shutdown is either permanent [21], represented by an infinite loop between Boxes 7 and 8, or temporary [9] where a core enters the same loop for a short period of time.

Unit-level reconfiguration: In this model (Figure 3), upon error detection, an error-discrimination technique similar to the previous scenario is applied. When an error is identified as intermittent, a fine-grained diagnosis technique is used to isolate the error-prone microarchitectural unit (Box 6). Once the defective unit is identified, a decision is made as to whether the unit must be shut down. If a core can operate without this unit (e.g., there exists a replica of this unit, or the program can be detoured to avoid using it [22]) then the unit is disabled and the core will continue running programs possibly with degraded throughput. In this case, the error discrimination variable for this core is reset. Similar to core-level reconfiguration, the duration of the unit shutdown is either permanent, represented by an infinite loop between

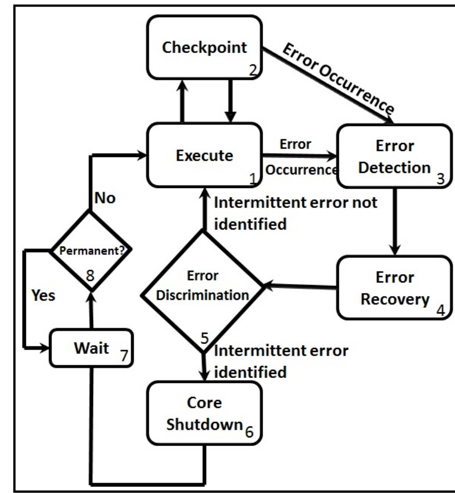

Fig. 2: An overview of an error-tolerant core with core shutdown.

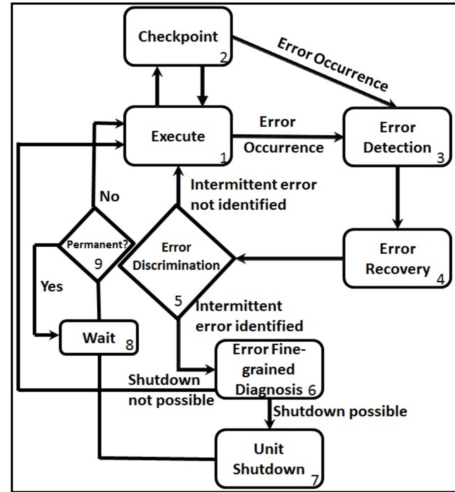

Fig. 3: An overview of an error-tolerant core with fine-grained reconfiguration.

Boxes 8 and 9, or temporary where a unit enters the same loop for a short period of time. Otherwise, the unit is *not* disabled and the core continues running the program despite the defective unit.

D. SAN models

We build the models discussed in the previous section using Stochastic Activity Networks (SAN). SAN's are a convenient, high-level, graphic abstraction for modelling stochastic systems. We use a single SAN in all our experiments. Different fault models and recovery scenarios are represented by changing parameters in this model. A prior knowledge of SANs is not required to understand our models.

Chip multiprocessor with unit-level reconfiguration and base fault model

Our model of a chip multiprocessor is depicted in Figure 4. When an application starts execution, it enters the *Run* place.

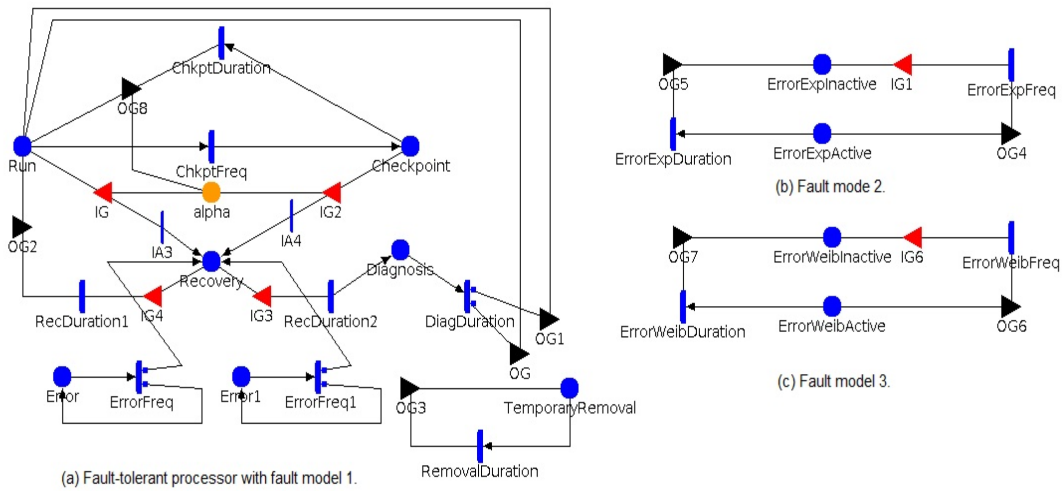

Fig. 4: Our SAN model for a fault-tolerant processor.

As the execution progresses, a checkpoint is taken periodically whenever *ChkptFreq* timed activity fires and then the program is transferred to the *Checkpoint* place where it stays there for a *ChkptDuration* duration. At every checkpoint the useful work is incremented at *OG8* output gate. The increment depends on how many cores and units are functioning in the processor at the checkpoint time.

At the same time, a token in the *Error* place models the errors that might occur because of the base fault model which follows exponential fault distribution with nominal MTTF. When the *ErrorFreq* timed activity fires, one of two possible outcomes (or cases) can take place. The top case models the probability of activated errors that either cause crashes (hardware trap or OS exception) or are detected through a software/hardware detector. The bottom case models: (1) inactivated faults, (2) activated faults that do not change the state of the program (benign faults) or (3) activated faults that change the state of the program but are not detected. When the top case executes, a token is moved from the *Error* place to the *Recovery* place. Both input gates *IG* and *IG2* ensure that whether the application (represented by a token) is in the *Run* or in the *Checkpoint* place, it will be transferred immediately to the *Recovery* place when *ErrorFreq* fires. Therefore, there are two tokens in the *Recovery* place at this point, one token from the *Error* place and another from either *Run* or *Checkpoint* place.

The *alpha* extended place is an array of alpha variables such that each core in the processor has its own alpha variable. A core's alpha is updated both on every checkpoint and on every crash/error detection to model the intermittent error discrimination mechanism in that core. One of the two tokens in the *Recovery* place can take one of the following directions:

(1) If the error is not identified as intermittent, then input gate *IG4* ensures that a token from the *Recovery* place is forwarded to *RecDuration1* timed activity which represents the time needed for an application to rollback to a checkpoint. The second token in *Recovery* place is not needed and will therefore be deleted. Then the output gate *OG2* places a token in the *Run* place which resembles the application resuming execution from the last checkpoint and losing all the work that has not been checkpointed.

(2) If the error is identified as intermittent, then a rollback to the last checkpoint is conducted through input gate *IG3* which ensures that a token from the *Recovery* place is forwarded to *RecDuration2* timed activity and then forwarded to the *Diagnosis* place. The second token in *Recovery* place is not needed and will therefore be deleted. The timed activity *RecDuration2* represents the time needed for an application to rollback to a checkpoint. The *DiagDuration* timed activity represents the time overhead of a diagnosis technique. Once this activity fires it can have one of two possible outcomes:

(1) Top case: the defective unit is diagnosed accurately. If the unit shutdown is temporary, then the disabled unit is removed from the set of available resources. In addition, a token is created in *TemporaryRemoval* place. This token stays there until *RemovalDuration* timed activity fires, this activity represents the duration of the shutdown (either temporary shutdown or permanent one, in which case the duration will be infinite). Once the timed activity *RemovalDuration* fires, the unit is put back into the pool of available resources. In addition, a token is put back in *Error1* place (*Error1* usage is explained in exponential and Weibull fault models) indicating that an error can happen in the future depending on the error distribution. The time required by the fine-reconfiguration technique is not modeled since this recovery action is invoked infrequently. Last, the alpha variable that corresponds to the reconfigured core is cleared upon diagnosis.

(2) Bottom case: the diagnosis process identifies a healthy unit as defective (inaccurate diagnosis). Therefore, a healthy unit is disabled, however, the intermittent error will still be active. In this case, the disabled unit is removed from the set of available resources until another diagnosis action is conducted. In addition, a token is created in *RemovalDuration* similar to the top case, but a token is put in the *Error1* place immediately.

In both cases, once the *DiagDuration* timed activity fires, a token is created in the *Run* place (by input gates *OG* and *OG1*), which resembles an application that resumes execution after a diagnosis and a potential reconfiguration. Further, a token is put back in *Error* which models the base fault model.

Rollback-only model: To build this model, input gate *IG3* is always disabled. Therefore, no fault discrimination, diagnosis or reconfiguration action is performed in this model.

Core-level reconfiguration model: To build this model, we do not need a diagnosis action since the array of alpha variables serves as a fault discrimination and diagnosis technique (remember that each core has its own alpha entry in the array). Therefore, the timed activity *DiagDuration* has a zero duration. Moreover, the bottom case in the same activity is always disabled.

Exponential and Weibull fault models: Exponential and Weibull fault models are built on the top of the model described earlier in this section.

Exponential fault model has fault arrivals that follow Poisson process (Figure 4 (b)). When the fault is inactive, a token is stored in place *ErrorExpInactive*, the input gate *IG1* ensures that the fault will not be activated if the defective core/unit is disabled. The timed activity *ErrorExpFreq* fires based on the corresponding MTTF, after which the output gate *OG4* creates a token in *Error1* place which represents the existence of a defective core/unit. In addition, a token is transferred from place *ErrorExpInactive* to place *ErrorExpActive*. Such token remains in the *ErrorExpActive* for the activity duration of the fault which is enforced by timed activity *ErrorExpDuration*. Once this duration has passed, the output gate *OG5* deletes the token in *Error1* place and the token from *ErrorExpActive* place is transferred back to place *ErrorExpInactive*.

Weibull fault model is very similar to exponential fault model (Figure 4 (c)). The main difference between the two models is that the activation times of intermittent faults in Weibull fault model are distributed according to Weibull distribution in timed activity *ErrorWeibFreq*.

TABLE II: Model parameters.

Parameter	**Value/Range**[1]	**Comments**
Base fault rate	Exponentially distributed with MTTF of 6.56 years	Found by Nightingale et al. [1].
Exponential fault model	Exponentially distributed with MTTF of 2 hours	-
Weibull fault model	Weibull distribution with MTTF of 1-40 hours	We study the sensitivity of our results to this parameter.
Fault rate during bursts	Exponentially distributed with MTTF of 5 seconds	-
Diagnosis duration	2 sec	Conservative duration since DIEBA reported millisecons overhead [18].
Recovery duration	0-60 sec	Conservative number since Wang et al. reported 10 minutes for the coordinated checkpoints [17].
Component rank	0%-35% of the processor's throughput	We study the sensitivity of our results to this parameter
Meant time to checkpoint (MTTC)	5-60 min	Found experimentally using user-level checkpoint in Linux [23]
Checkpoint duration	30 sec	Found experimentally using user-level checkpoint in Linux [23]
Probability that an intermittent fault is activated	0.75	Found using microarchitecture simulator [19].
Probability that an error is detected	0.7	Conservative number for detection coverage [24]
Probability of crash due to an intermittent fault	0.53	Found using microarchitecture simulator [19].
Probability of benign intermittent faults	0.34	Found using microarchitecture simulator [19].
Probability of silent-data corruption due to an intermittent fault	0.13	Found using microarchitecture simulator [19].
Number of on-chip cores	2 to 32 cores	-

IV. Experiment Setup

We use the Mobius modeling framework to construct and simulate our SAN models [5]. We evaluate the useful work of the model described in the previous section after 48 hours from the intermittent fault occurrence using Mobius simulations with a confidence level of 95%. The parameters we use are shown in Table II.

In our experiments, we focus on finding answers to the following research questions:

(1) When should we recover from an intermittent fault by a simple program rollback to a checkpoint and when should we shutdown the defective component?

Typically, to recover from a hardware fault, the first step is to identify the type of the fault (permanent, transient or intermittent). In our work, we show that to recover from an intermittent fault we also need to identify the error rate and importance of the defective part. A fault that occurs frequently in a critical processor component, for example, might be tolerated by a simple program rollback. An error that occurs frequently in a part that has a replica, on the other hand, should be tolerated by disabling that faulty part.

(2) For the errors that are tolerated by shutting down the defective component, should the shutdown be permanent or temporary?

An alternative fault mitigation approach that targets intermittent-fault prone processors is to temporarily shutdown the defective parts of the processor, to relieve such parts from the stressful operating conditions [9]. For example, if the processor is experiencing NBTI-related failure, pausing or migrating any workload that is currently running on the processor might reduce the processor's temperature and hence the processor can "partially" self-recover from the failure (Section II-B). However, it is unclear which error rates can be

tolerated by temporary shutdown of the defective components.

(3) What is the granularity of the disabled component that maximizes the processor's performance?

Traditionally, a processor that is diagnosed with a hardware error would be replaced with another hot or cold processor (hot/cold swapping) [9], [21]. Another recovery approach is to reconfigure the defective processor and facilitate "graceful degradation" where *only* the defective core or microarchitectural structure is shutdown [12], [22], [25]–[27]. In the latter approach, the remaining operating parts of the processor can be used after the shutdown of the defective part, hence, a processor would function with degraded performance. We show that the granularity of the error location (in addition to the error severity) is an important factor in choosing the right recovery action.

To evaluate the relative importance of the disabled part of the processor, we propose a new metric that we call *component rank*. We use the term component to refer to a microarchitectural unit or a core in a processor. **Component rank** is the maximum percentage of processor useful work that is lost when the corresponding component is disabled upon error recovery. For every component in the processor, we can calculate its rank as follows:

1) Components are assigned a rank of 0% if they (a) have a replication component which is rarely used or (b) can be replaced or repaired using low-cost techniques.
2) Components are assigned a rank of 100% if they are critical for the workload executed by the processor such that the workload cannot be executed without such component.
3) The rest of the components are assigned ranks that depend on how much useful work will be lost if such component is disabled. To estimate such loss, one can use Amdahl's law[2], which states that the speedup (or slowdown, in our case) of a program depends on how much of this program is improved (or downgraded, in our case).

To evaluate the useful work, we divide the work that can be done by the reconfigured processor (the work lost in a reconfigured processor depends on the rank of the disabled component) by the work that can be done by a healthy processor with no disabled parts.

V. RESULTS

In our first set of experiments, we run two recovery scenarios (rollback-only, unit-level reconfiguration with both temporary permanent shutdown) for the three fault models described in Section II. In this experiment, we set the MTTC to 20 minutes, recovery duration to 30 seconds, MTTF for exponential fault model and Weibull fault model to 2 hours and component rank to 5%. The diagnosis accuracy is 100% and diagnosis duration is 2 seconds. We assume that the recovery technique can shutdown the defective component upon successful diagnosis.

[2]Quantitatively, Amdahl's law states that if a part p of a program is sped up by s then the program overall speed up is $1/((1-p)+p/s)$.

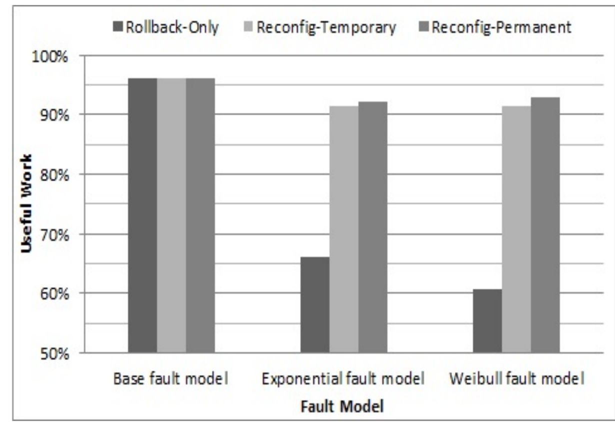

Fig. 5: Useful work for the different recovery scenarios using three fault models.

We plot the useful work in Figure 5. In the following discussion, we explain the results obtained from this figure in form of answers to the first two research questions described in Section IV.

(1) When should we recover from an intermittent fault by a simple program roll back to a checkpoint and when should we shutdown the defective component?

The rollback-only recovery performs well for the base fault model in which there are no bursts of extreme failure rates. The only lost useful work (about 4%) is due to time spent to save checkpoints. However, the rollback-only recovery performs poorly for exponential fault model and Weibull fault model in which there are bursts of extreme failure rates. The useful work drops to about 63% for both fault models, on average. On the other hand, temporary or permanent reconfiguration of the defective component produces more useful work than rollback-only recovery by 25% for exponential fault model and 30% for Weibull fault model.

(2) For errors that are tolerated by shutting down the defective component, should the shutdown be permanent or temporary?

Temporary and permanent reconfiguration have close gains in terms of useful work. However, permanent shutdown of the defective component achieves 2% more useful work than temporary shutdown. This difference in performance of the permanent and temporary shutdown is the result of the overhead of the fault distinguishing and restore-to-last-checkpoint mechanisms that are continuously encountered for each error burst. Such overhead is encountered only once by the permanent shutdown scenario.

In the next experiment, we evaluate the impact of the relative importance of the disabled component on recovery by varying the rank of the defective component (see Section IV for more details about component rank). A low component rank may represent a microarchitectural unit, while a high component rank may represent a core in a processor or a critical microarchitectural unit.

We use the permanent shutdown recovery scenario since it

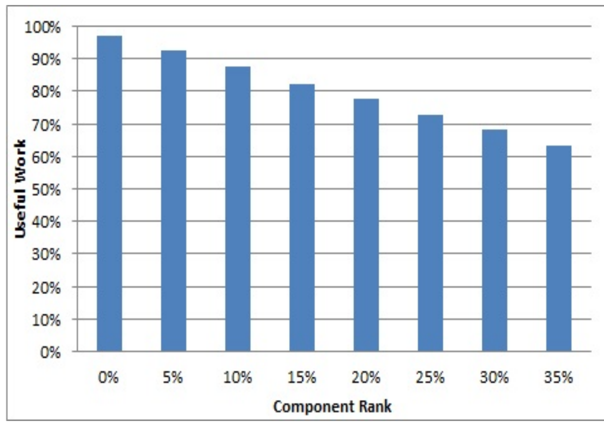

Fig. 6: Useful work for different ranks when permanently reconfiguring a defective component.

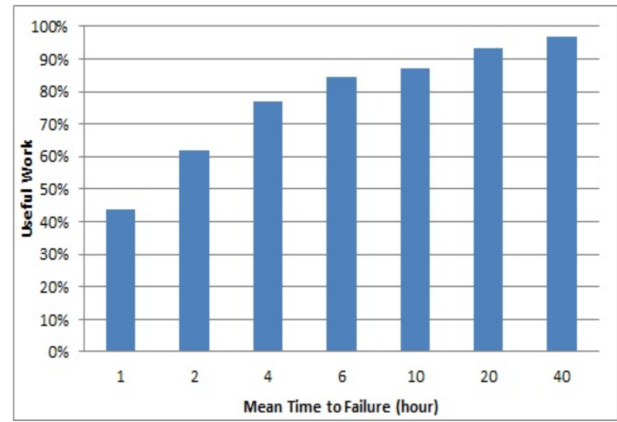

Fig. 7: Useful work for the rollback-only recovery using Weibull fault model with variable MTTF.

generates the highest percentage of useful work in the previous experiment. The rest of the experimental setup is similar to the previous experiment.

By analyzing the useful work generated by a processor after permanently shutting down a defective component with different ranks, we answer the following research question:

(3) What is the granularity of the disabled component that maximizes the processor's performance?

The smaller the granularity of the disabled component, the more useful work that is achieved after processor reconfiguration. This can be done by disabling the defective microarchitectural unit rather than disabling the entire defective core. For this particular experiment, we find that disabling a component with a rank of 35% or more in the permanent reconfiguration recovery scenario has similar effects on useful work to the rollback-only recovery technique in which the defective component is used together with functioning components. We illustrate this result with some real-world examples.

Example 1: A 4-core processor is used to run floating point-intensive application. This processor has one floating point unit (FPU). Assume that an intermittent fault (that follows Weibull fault model) affects the FPU. The FPU has a rank of 100% as the processor is deemed unusable upon FPU shutdown. Therefore, a rollback-only recovery scenario is the most beneficial recovery in this case.

Example 2: A 4-core processor in which each core has two load-store units (LSUs) is running a highly parallel memory-intensive program that is using all the 8 LSUs for 60% of the time. An intermittent error that follows Weibull fault model with an MTTF of 2 hours affects one of the LSUs. According to Amdahl's law, the useful work will be degraded by 19% or $1/(0.4 + (0.6/0.125))$ upon the defective LSU shutdown. Hence, the component rank in this case is 19%. Based on Figure 6, since 19% is less than the threshold of 35% for such intermittent fault rate, the best recovery option for this case is a permanent reconfiguration.

Example 3: Assume that the same processor described in the previous example is used to run a memory-intensive application that uses 4 LSUs for 60% of the time. Moreover, suppose that the processor experiences an intermittent error that affects one of its LSUs. It is highly unlikely that shutting down such unit will cause any loss of the useful work. Therefore, the component rank in this case is 0% and a permanent reconfiguration recovery scenario is recommended.

A. Sensitivity analysis

In this section, we study the sensitivity of our results to MTTF, MTTC and recovery duration for the rollback-only recovery.

Unless otherwise mentioned, the experimental settings are as follows: Weibull fault model is used with MTTF of 2 hours. MTTC is 20 minutes, the recovery duration is 30 seconds and the diagnosis accuracy is 100%.

Mean Time To Failure: In this analysis, we evaluate the useful work for the rollback-only recovery. MTTF in this experiment is varied between 1 to 40 hours (Figure 7). Intuitively, the more frequent the error becomes, the less useful work that is accomplished. This is due to the time needed by the recovery and the loss of computations that are not checkpointed. *As a general rule, if the intermittent error is frequent enough such that its effect on the useful work outweighs the rank of the defective component, then such component should be disabled. Otherwise, the system can continue exploiting the stored checkpoints to recover from the infrequent failures.*

Mean Time To Checkpoint: We now plot the useful work for the rollback-only recovery with variable mean time to checkpoint (Figure 8). Although storing a checkpoint every one hour reduces the overhead of checkpointing, it degrades the accomplished useful work. This is because more work is lost when a failure occurs. Moreover, although reducing the MTTC decreases the overhead for each checkpoint due to less amount of data being stored, there is a fixed overhead of each checkpoint that results from the context switches between the running program and the checkpoint mechanism. In our experiments, we rely on previous work to choose an MTTC of

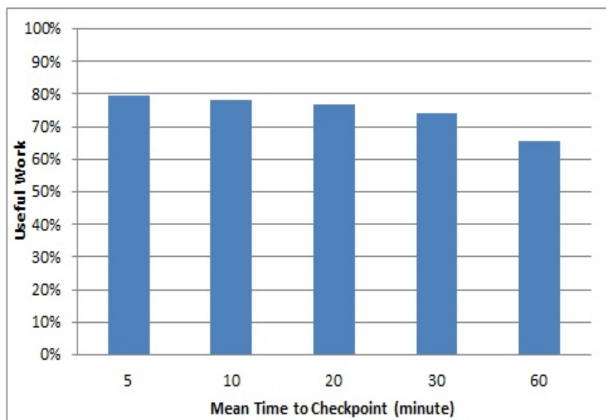

Fig. 8: Useful work for the rollback-only recovery using Weibull fault model and variable MTTCheckpoint.

20 minutes, which serves as a balance between the checkpoint frequency and the overhead associated with the mechanism.

Therefore, if a processor is encountering frequent bursts of intermittent errors and reconfiguration is not possible, then one can reduce the mean time to checkpoint to increase the processor useful work.

Recovery duration: In this experiment, we evaluate rollback-only model with variable mean time to recover. We measure the useful work (figure not shown due to space constraints) of the processor. We find that the useful work is only marginally affected by the recovery duration.

VI. Discussion

To shed some light on the applicability of our model to real processors, we discuss the assumptions we made in our model and the impact of these assumptions on the results we obtain from our study. The assumptions are as follows:

(1) The model does not take into account correlated faults, or faults that appear in units that form a "hot spot" in the processor, for example. Note that hot spots that consist of multiple close by microarchitectural units usually occur much faster than chip-wide heating [28]. There are multiple factors that result in an intermittent fault triggering diagnosis and recovery techniques: (a) the error has to occur despite the power and temperature management techniques (e.g. HotSpot [28]), (b) the error must manifest itself to the program and bypass the architectural masking, such as two timing errors cancelling each other, (c) the error must propagate to the program, cause erroneous behavior and bypass the instruction-level masking, (d) the error must happen frequently enough to be distinguished as intermittent and (e) the error must happen frequently enough that it triggers a reconfiguration technique. In our analysis, we model factors c, d and e, but we do not model factors a and b (as we assume that only one microarchitectural unit is affected by an error). If multiple units expose an intermittent error to the program, then this means that more advanced error discrimination and diagnosis techniques should be applied in our model. Such techniques may impose more performance overhead and degrade the useful work before reconfiguring the core. Moreover, disabling two or more microarchitectural units may result in the core being unable to function properly so core shutdown may be more useful than unit shutdown in this case.

(2) We do not model errors that occur during program rollback to the last checkpoint. Errors that occur during this stage can be detected through error detectors; when an error is detected the program rollback can start over (assuming that the error will eventually stop and the program will not enter into endless loop of error detection and restarting). The effect of this process will be a longer rollover.

(3) We model the effects of inaccurate error diagnosis/reconfiguration by modeling the inaccuracies in diagnosis results (this can represent imperfect diagnosis technique or errors that affect a diagnosis/reconfiguration technique). In our model, we assume that after an incorrect diagnosis/reconfiguration the error will persist, therefore, the disabled unit will be enabled and error tolerance techniques will be applied again.

VII. Related Work

The fault recovery scenarios we study are inspired by recovery proposals in literature. For example, Schuchman et al. [25] (Rescue) and Shivakumar et al. [29] proposed to remedy permanent faults by exploiting redundancy at the microarchitectural-level to compensate for the disabled-defective units. They both targeted hard faults. Romanescu et al. [12] proposed core cannibalization in which cores that are identified to be permanent-fault-prone are cannibalized to pipeline-stage parts. Then the functional parts of a defective core serve as a "supply of replacements" by other fault-free cores. They do not consider the impact of error detection, diagnosis, recovery and reconfiguration on the performance of the chip. Powell et al. [26] proposed core salvaging, in which the defective units in a core are disabled. The defective core falls into other healthy cores whenever the disabled units are needed. Meixner el al. [22] proposed to disable the defective units and to use application detouring. Detouring is to convert instructions that use the defective units to equivalent instructions that use healthy units only, where possible. This conversion is done using binary translation layers. Both [26] and [22] target permanent faults. Wells et al. [9] proposed to recover from intermittent errors by disabling the core and letting it cool down for sometime. Their rationale is that some intermittent faults might be induced by fluctuating temperature and voltage, and hence if a core is shutdown for a while then its temperature and voltage would stabilize.

Intermittent faults, in particular, aging faults can be attributed to gate-oxide breakdown, NBTI, hot-carrier injections and electromigration. Srinivasan et al. [13] used RAMP [30] to model two recovery scenarios for aging-related errors. The first scenario consists of spare microarchitectural redundancy that will only be used when other units fail. The second recovery scenario involves using the existing (not spare) microarchitectural redundancy within a core and degrading performance gracefully as units fail. They also consider hybrid models of

both scenarios. Our work is different in that: (1) we build an end-to-end model that includes error discrimination, detection, diagnosis and recovery while they focused on error recovery only and (2) we add a model that suspends the offending part of the processor temporarily or permanently while they focused on permanent shutdown of the processor units only.

Bondavalli et al. [16] propose a count-and-threshold-based model to discriminate between intermittent and transient faults. Their model assumed Weibull distributed intermittent faults and bursty transient-fault intervals. Later, the authors extend their fault-discrimination model by adding models for fault diagnosis and recovery for nodes (a number of processors) in distributed systems [15]. They assume one recovery scenario for intermittent faults, which is to remove the defective node. In this work, we build on [16] by exploiting their count-and-threshold model, however, we consider different models for diagnosis and recovery models at the processor level.

VIII. Conclusions

Since intermittent errors are emerging as one of the leading causes of hardware failures, there is a critical need to design optimal recovery actions for such errors. We modeled a fault-tolerant chip multiprocessor that experiences intermittent hardware faults during its lifetime. We also modeled intermittent faults at a high level by relying on insights from related work at the physical level. We evaluated the chip performance for different recovery scenarios under different intermittent fault models and rates. Although it suffices to know the error type only to choose the right recovery action for transient or permanent hardware faults, we find that the failure rate and error location are also important factors in choosing the right recovery action for intermittent errors.

Acknowledgements

This work was supported in part by Discovery and Engage grants from the National Science and Research Council of Canada (NSERC). We thank the anonymous reviewers of QEST 2012 for suggestions that helped to improve this work.

References

[1] E. Nightingale, J. Douceur, and V. Orgovan, “Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer pcs,” *European Conf. on Computer Systems*, 2011.

[2] C. Constantinescu, “Intermittent faults and effects on reliability of integrated circuits,” *Reliability and Maintainability Symp.*, pp. 370–374, 2008.

[3] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, and A. Keshavarzi, “Parameter variations and impact on circuits and microarchitecture,” *Design Automation Conf.*, pp. 338–342, 2003.

[4] J. McPherson, “Reliability challenges for 45nm and beyond,” *Design Automation Conf.*, pp. 176–181, 2006.

[5] T. Courtney, D. Daly, S. Derisavi, V. Lam, and W. Sanders, “The mobius modeling environment,” *Tools of the Intl Multiconference on Measurement, Modelling and Evaluation of Computer Communication Systems*, no. 781, 2003.

[6] M. Depas, M. Heyns, and P. Mertens, “Soft breakdown of ultra-thin gate oxide layers,” *Proc. of the European Solid State Device Research Conf.*, vol. 25, pp. 235 – 238, 1995.

[7] V. Reddy, A. Krishnan, A. Marshall, J. Rodriguesz, S. Natarajan, T. Rost, and S. Krishnan, “Impact of negative bias temperature instability on digital circuit reliability,” *Reliability Physics Symposium*, pp. 248 – 254, 2002.

[8] Y. Huang, T. Yew, W. Wang, Y.-H. Lee, R. Ranjan, N. Jha, P. Liao, J. Shih, and K. Wu, “Re-investigation of gate oxide breakdown on logic circuit reliability,” *Reliability Physics Symp.*, pp. 2A.4.1 – 2A.4.6, 2011.

[9] P. Wells, K. Chakraborty, and G. Sohi, “Adapting to intermittent faults in multicore systems,” *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 255–264, 2008.

[10] H. K. S. W. S. M. e. a. M. Ershov, S. Saxena, “Dynamic recovery of negative bias temperature instability in p-type metaloxidesemiconductor field-effect transistors,” *Appl. Phys. Lett*, vol. 83, no. 8, 2003.

[11] C. Weaver and T. Austin, “A fault tolerant approach to microprocessor design,” *Int. Conf. on Dependable Systems and Networks*, pp. 411 – 420, 2001.

[12] B. Romanescu and D. Sorin, “Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults,” *Intl. Conf. on Parallel Architectures and Compiliation*, pp. 43–51, 2008.

[13] P. B. J. Srinivasan, S.V. Adve and J. Rivers, “Exploiting structural duplication for lifetime reliability enhancement,” *Intl. Symp. on Computer Architecture*, vol. 33, 2005.

[14] J. Blome, S. Feng, S. Gupta, and S. Mahlke, “Self-calibrating online wearout detection,” *Int. Symp. on Microarchitecture*, 2007.

[15] M. Serafini, A. Bondavalli, and N. Suri, “Online diagnosis and recovery: On the choice and impact of tuning parameters,” *IEEE Transactions on dependable and secure computing*, vol. 4, no. 4, p. 2007, 2007.

[16] A. Bondavalli, S. Chiaradonna, F. Giandomenico, and F. Grandoni, “Threshold-based mechanisms to discriminate transient from intermittent faults,” *IEEE TRANSACTIONS ON COMPUTERS*, vol. 49, no. 3, 2000.

[17] L. Wang, K. Pattabiraman, Z. Kalbarczyk, R. Iyer., L. Votta, C. Vick, and A. Wood, “Modeling coordinated checkpointing for large-scale supercomputers,” *Int. Conf. on Dependable Systems and Networks*, pp. 812 – 821, 2005.

[18] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, “Dieba: Diagnosing intermittent errors by backtracing application failures,” *Silicon Errors in Logic - System Effects*, 2012.

[19] L.Rashid, K.Pattabiraman, and S.Gopalakrishnan, “Characterizing the impact of intermittent hardware faults on programs,” *Technical Report, UBC*, 2012.

[20] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, “Modeling the propagation of intermittent hardware faults in programs,” *Pacific Rim Intl. Symposium on Dependable Computing*, pp. 19–26, 2010.

[21] I. Parulkar, T. Ziaja, R. Pendurkar, A. D’Souza, and A. Majumdar, “A scalable, low cost design-for-test architecture for ultrasparc chip multi-processors,” *Int. Test Conference*, pp. 726 – 735, 2002.

[22] A. Meixner and D. Sorin, “Detouring: Translating software to circumvent hard faults in simple cores,” *Proc. of the Intl. Conf. on Dependable Systems and Networks*, pp. 80–89, 2008.

[23] J. Plank, M. Beck, G. Kingsley, and K. Li, “Libckpt: Transparent checkpointing under unix,” *Usenix Winter Technical Conference*, pp. 213–223, 1995.

[24] K. Pattabiraman, G. Saggese, D. Chen, Z. Kalbarczyk, and R. Iyer, “Automated derivation of application-specific error detectors using dynamic analysis,” *IEEE Transactions on Dependable and Secure Computing*, vol. 8, 2011.

[25] E. Schuchman and T. Vijaykumar, “Rescue: A microarchitecture for testability and defect tolerance,” *Intl. Symp. on Computer Architecture*, pp. 160–171, 2005.

[26] M. Powell, S. Gupta, and S. Mukherjee, “Architectural core salvaging in a multi-core processor for hard-error tolerance,” *Intl. Symp. on Computer Architecture*, vol. 37, 2009.

[27] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke, “The stagenet fabric for constructing resilient multicore systems,” *Intl. Symp. on Microarchitecture*, pp. 141 – 151, 2008.

[28] K. Skadron, M. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, “Temperature-aware microarchitecture: Modeling and implementation,” *ACM Transactions on Architecture and Code Optimization*, vol. 1, no. 1, pp. 94–125, 2004.

[29] P. Shivakumar, S. W. Keckler, C. R. Moore, , and D. Burger, “Exploiting microarchitectural redundancy for defect tolerance,” *Intl.Conf. on Computer Design*, pp. 481 – 488, 2003.

[30] J. Srinivasan, S. Adve, P. Bose, and J. Rivers, “The case for lifetime reliability-aware microprocessors,” *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2, 2004.