Fault Detection and Recovery Efficiency Co-optimization through Compile-time Analysis and Runtime Adaptation

Hao Chen and Chengmo Yang
Department of Electrical and Computer Engineering
University of Delaware
140 Evans Hall, Newark, DE 19716
{hchen, chengmo}@udel.edu

***Abstract*—The ever scaling-down feature size and noise margin keep elevating hardware failure rates, requiring the incorporation of fault tolerance into computer systems. One fault tolerance scheme that receives a lot of research attention is redundant execution. However, existing solutions are developed under the assumption that the fault rate is low. These techniques either solely focus on fault detection, or sometimes even increase recovery cost to reduce fault detection overhead. The lack of overall efficiency makes them insufficient and inappropriate for embedded systems with tight energy and cost budget.**

Our study shows that checkpoint frequency and fault rate are two critical parameters determining the overall fault detection and recovery overhead. To co-optimize detection and recovery, we statically construct a mathematical model, capable of taking application and architecture characteristics into consideration and identifying the optimal checkpoint frequency of an application for a given fault rate. Moreover, as the fault rate is infeasible to predict a priori, we furthermore propose a set of heuristics, enabling the system to dynamically monitor the fault rate and adapt the checkpoint frequency accordingly. The efficacy of the static and the adaptive optimizations is evaluated through detailed instruction-level simulation. The results show that the optimal checkpoint frequency identified by the static model is very close to the actual value (6% deviation) and the run-time adaptation scheme effectively reduces the overhead caused by the unpredictability in fault rate.

I. INTRODUCTION

The unending quest for producing embedded systems with higher performance and lower power is challenged by the increasing susceptibility of silicon devices to hardware defects. The fault rate has increased by three orders of magnitude (from 10^{-7} to 10^{-4}) [1], [2] as technology advanced from 180nm to 45nm. With devices forecasted to shrink down below 18nm in scale by 2015 [3], issues such as Soft-Breakdowns (SBD), Negative Bias Temperature Instability (NBTI), Electro-Migration (EM), and dielectric breakdown [4], will become clear threats for systems in near future technologies. As computation advances towards exascale and devices advance towards nanoscale, faults are expected to occur in a *continuous* manner in future systems, across *all* levels from hardware to application.

Along with the elevation in fault rate, we also expect a variation in fault duration. In addition to transient faults and permanent faults, intermittent faults may occur frequently and irregularly during application execution. These faults are commonly due to, among other factors, process variation or in-progress wear-out combined with voltage and temperature fluctuations [5], [6]. Together these factors would cause fault duration to vary over nanosecond to second time scales.

The severe reliability challenges, along with the problem of computer systems becoming too large to be functionally tested at the manufacturing exit, impose stringent requirements on the incorporation of fault tolerance into future designs. A typical fault tolerant system provides three types of functions: *fault detection*, *execution checkpointing*, and *error recovery*. These three functions are strongly interrelated such that optimizing one of them may end up adversely affecting the other two. As an illustrative example, consider a program that computes the sum of a set of numbers. Assume that the program accumulates a partial sum and adds numbers one by one. Also assume that this program is duplicated into two threads for redundant execution. The cheapest fault detection approach is to compare only the final sum. Yet a mismatch of the two final sums would require both threads to be rolled back to the beginning for re-execution, since none of the intermediate partial sums are checked or recorded. In contrast, comparing and checkpointing every partial sum minimizes error recovery overhead, but constantly imposes a fair amount of comparison and storage overhead regardless of whether or not a fault occurs.

The example above illustrates the fact that a more coarse-grained fault detection/checkpointing approach usually causes the recovery overhead to increase. Previously when the fault rate was low, recovery was not a critical concern. As a result, existing reliability techniques either solely focus on fault detection [7], [8], or sometimes even increase recovery overhead [9], [10], [11] or sacrifice fault coverage [12], [13] to reduce fault detection overhead. The lack of overall efficiency makes these solutions insufficient for future systems where faults are expected to occur in a continuous manner.

The projected high fault rate in future systems requires a re-examination of the checkpointing strategy so as to optimize the overall efficiency of fault detection and recovery. As application characteristics may have significant impact on fault propagation, it is necessary to develop an accurate model capable of capturing such impact on fault detection and recovery overhead. Moreover, as the exact runtime fault rate is usually unpredictable, *adaptivity* needs to be incorporated into the checkpointing process, so that the fault detection and recovery overhead can be dynamically balanced to match the fault rate.

In this paper, we present our work in developing an integrated and adaptive fault detection, checkpoint, and recovery framework. We select redundant execution, the most widely adopted fault tolerance approach as the baseline scheme. To achieve maximum efficiency in the face of high and unpredictable fault rates, both compile-time and runtime optimizations are exploited. The major contributions of this paper are outlined as follows:

978-1-4799-1400-5/13/$31.00 ©2013 IEEE

- We conduct a thorough and concrete mathematical analysis to identify the factors that determine the overall efficiency of fault detection and recovery. Based on application and architecture characteristics extracted at compile-time, a mathematical model of checkpoint frequency is constructed.
- We introduce two runtime heuristics to monitor the fault rate and then adapt the checkpoint frequency accordingly. Both heuristics can be efficiently implemented with minimal hardware support.
- We develop a detailed, instruction-level fault injection and simulation framework to evaluate the efficacy of the compile-time model and the runtime heuristics. Extremely high fault rates are adopted to mimic the fault behavior in future systems.

The rest of this paper is organized as follows. Section II briefly reviews current fault tolerance techniques and their limitations, while Section III outlines the technical motivation. Section IV introduces the details of our fault tolerance framework and the corresponding mathematical model, while Section V presents the proposed runtime heuristics. Section VI outlines the fault injection and simulation framework as well as the results of our experimental studies. Section VII summarizes the paper.

II. Technical Background

The elevation in fault rates has caused increasing research attention to be paid to the incorporation of reliability support into computer systems. For embedded systems with tight power and resource constraints, the need for highly efficient fault resilience methods becomes increasingly critical and urgent. It is thus necessary to evaluate a technique not only by its *effectiveness* in detecting faults and recovering the affected computation, but more importantly by its *efficiency* in terms of the associated performance, energy and hardware overhead.

A. Fault Detection and Recovery

Fault tolerance requires some form of redundancy. The degree of the required redundancy increases for components that lack a regular structure. For storage components and communication channels with regular structures, data can be protected by Error Correcting Codes (ECC) and parity bits. In contrast, logic and computation blocks, which typically have irregular structures, require greater redundancy so *redundant execution* is usually considered as the best approach for tolerating arbitrary faults.

Compared to error coding techniques that protect each hardware structure individually, redundant execution can provide greater fault coverage across the entire system. Researchers have performed intensive studies on redundant execution. AR-SMT [14] adopted simultaneous multithreading (SMT) to redundantly execute two threads. It compares all the instructions, and delays result committing until these results have been verified. SRT [8] reduced overhead of AR-SMT by maintaining a slack between the two redundant threads to reduce branch misprediction and cache misses, and only comparing store instructions. The same idea was later applied to chip multiprocessors (CMPs) in [7].

To reduce the comparison overhead of redundant execution, dependence-based checking elision (DBCE) [11] was proposed to dynamically construct dependence chains of instructions, and only check the last instruction in each chain. Another set of techniques turn off one thread in high performance regions [12], [13]. They sizably reduce duplication overhead, however, at the cost of significantly increased rates of undetectable faults.

When a fault is detected, recovery can be performed, either by preventing faults from modifying system states, or by rolling the computation back to a previously saved clean state – a *checkpoint*. Systems that prevent faults from modifying computation states can recover from a fault in the same way as they recover from incorrect speculation. Techniques of this category include SRTR [11] for SMT processors and CRTR [9] for CMPs. However, these techniques need to buffer and check each instruction result.

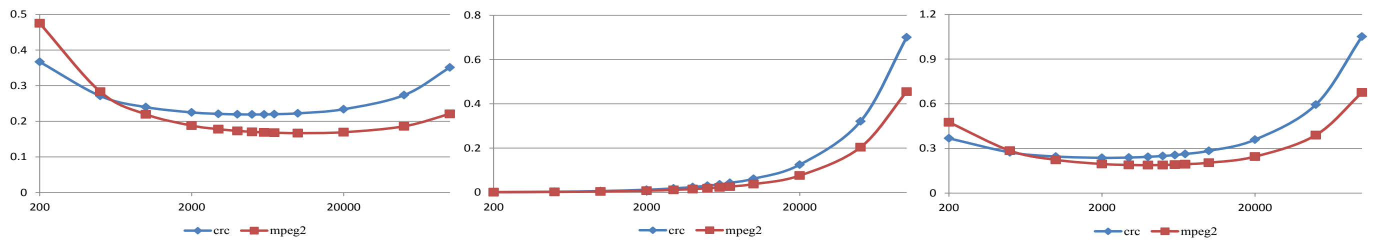
Alternatively, a system can allow unconfirmed results to be written into the registers, and only compares store values and addresses for fault detection. Techniques of this category include the RVQ-free recovery in [15] and the cache-based checkpointing in [16]. To ensure the existence of a clean state upon a detected fault, these techniques need to periodically save the processor state (including program counter, register file, and other hardware registers) and verify its correctness to ensure that a newly created checkpoint is fault-free. The higher the fault rate, the more frequently the computation need to be checkpointed.

B. Checkpoint optimizations

There are typically two ways of generating a checkpoint: *incremental* and *periodic*. In incremental checkpointing [10], the system keeps a log of new values as they are generated. On detecting a fault, this log is traversed to regenerate the exact system state when the fault was detected. The overhead to create the checkpoint is minimal, but the time to recover the computation is significant. In periodic checkpointing [17], the processor periodically takes a clean snapshot of its state. This scheme simplifies the recovery process yet imposes non-trivial checkpointing overhead because not only does a large amount of state need to be copied, but a large number of values need to be compared to ensure that the newly created checkpoint is fault free. To reduce the comparison overhead, one way [18] is to capture all updates to the architectural state in a global running fingerprint and only compare the fingerprint prior to generating a checkpoint. Another approach [19] is to only compare and checkpoint a minimal set of registers at each checkpoint. The technique guarantees that the registers that are not selected for checkpointing will never influence program final results.

Previous studies on optimizing checkpoint frequency focus on theoretical analysis. In [20], a first order approximation was proposed to identify the checkpoint interval that minimizes the cost of failures. This work was later improved in [21] with a higher order approximation and the consideration of more factors such as multiple failures between checkpoints and the failure during the restart process. In [22], a mathematical model was proposed for real time systems to maximize the probability that a task can finish before its deadline. A constant checkpoint interval is determined according to the checkpoint overhead and the constant failure rate. An approach based on the calculus of variations theory was proposed in [23] to minimize the expected cost of checkpoint and recovery. The same theory was later utilized in [24] to model the checkpoint placement problem with both infinite and finite time horizons. A more recent work in [25] presented a checkpoint/restart model, and solved the checkpoint problem assuming a stochastic renewal reward process.

As the statically constructed theocratical models are not able to capture runtime system behavior, researchers started to develop adaptive checkpointing schemes. An online-algorithm was proposed in [26]. Runtime information is exploited to identify positions at which the system state size is small, and checkpoints



(a) Fault Detection Overhead (b) Recovery Overhead (c) Overall Overhead

Fig. 1. **Fault detection and recovery overhead as a function of checkpoint interval.** The horizontal axis represents the checkpoint interval and the vertical axis represents the overhead. These data are collected under a failure rate of 10^{-5} per instruction.

are only placed in such positions. The concept of cooperative checkpointing was introduced in [27]. The programmer statically inserts checkpoints in program, while the system dynamically grants or denies a checkpoint based on runtime information. An adaptive scheme for realtime systems was proposed in [28]. It improves the mathematical model proposed in [22] by dynamically adjusting the checkpoint frequency according to parameters such as the remaining execution time of the program and time left before deadline, thus maximizing the probability that the current job can meet its deadline. In [29], the MeanFailureCP scheme was proposed for distributed systems. It dynamically adjusts a job's checkpointing interval as a function of the remaining job execution length and the specific failure interval of the node that the job is executed on. This work was later extended in [30] in a way that adapts the checkpoint interval in the absence of the knowledge of job execution length.

III. Technique Motivation

The trend towards elevated and clustered fault behavior mandates a transformation in fault tolerance approaches. Previously when the fault rate was low, recovery was not a critical concern. Thus more research attention was paid to reducing the overhead of fault detection. However, in the future when faults occur in a continuous manner, recovery overhead will become as critical as detection overhead. Moreover, as applications become complex and lengthy, it is necessary to periodically checkpoint the computation so that upon detecting a fault, the system does not have to re-execute the entire program.

The overall efficiency of fault detection and recovery is highly influenced by the *checkpointing strategy*. To illustrate this impact, let us consider a scheme that detects faults through redundantly executing two threads and comparing the results written to the memory (i.e., the address and value of each store instruction) and the value of each register when reaching a checkpoint. Upon any mismatch, a fault is detected, and the computation is rolled back to the most recent checkpoint. Figure 1 plots the fault detection and recovery overhead of two programs, *mpeg2* and *crc*, as a function of the checkpoint interval. The number of comparisons performed per instruction is reported as the fault detection overhead (Figure 1a), while the ratio of extra instructions executed during the recovery process is reported as the recovery overhead (Figure 1b). It can be clearly seen that as the size of checkpoint interval increases, the detection overhead decreases[1] since the register values are compared less frequently, while the recovery overhead increases since the average distance to the latest checkpoint becomes larger. The overall efficiency (i.e., the sum of the two types of overhead) is therefore determined by the checkpoint frequency.

While the data in Figure 1 are collected under the same fault rate and using the same detection and recovery scheme, the two programs still exhibit sizable difference in both the detection and the recovery overhead. This reflects the impact of *application characteristics*. Fault detection overhead varies across applications because store values and addresses need to be compared, while the ratio of store instructions varies significantly across applications. Recovery overhead also varies across applications because recovery is performed upon detecting a fault, while the ratio of detected faults to all the injected faults is highly influenced by the control and data flow of an application. During execution many faults will be masked, such as the faults occurring in dynamically dead instructions[2], or in registers that are not live[3].

Not only application characteristics, but also runtime adaptivity needs to be considered. As the fault behavior will be highly influenced by voltage and power during application execution, the system needs to monitor the fault behavior and adapt the checkpoint frequency accordingly so as to balance the overall overhead of fault detection and recovery. The challenge, however, is to efficiently implement the monitoring and the adaptation mechanisms in hardware so as to minimize the associated timing and energy overhead. Minimizing the power overhead ensures that these extra functions will hardly influence the system's fault behavior, while minimizing the turnaround time is crucial to the steadiness of the system conditions that these functions adapt for.

Motivated by these observations, our goal is to develop an integrated fault detection, checkpointing, and recovery framework, and furthermore to adaptively tune the granularity of these functions to match the fault rate. The overall framework, shown in Figure 2, is composed of two parts, namely, the static model and the dynamic heuristics.

- The static model determines the static-optimal checkpoint (CKPT) frequency based on the predicted fault rate. Unlike previous static models [20], [23], [28] that simply assume a constant overhead per checkpoint interval, our model will take application-specific information into consideration, including the ratio of detected faults, the instruction distribution, and the exact checkpoint overhead.
- The dynamic heuristics monitor the real fault behavior and adjust the runtime CKPT frequency accordingly. Unlike

[1]Except when the checkpoint interval is very large and the comparison of stores dominates fault detection. The detection overhead increases since more stores need to be executed during recovery.

[2]Dynamically dead instructions are those whose values are either not used by any instruction or only used by other dynamically dead instructions.

[3]At a given time point, a register is *live* if the very first subsequent access to that register is a read operation.

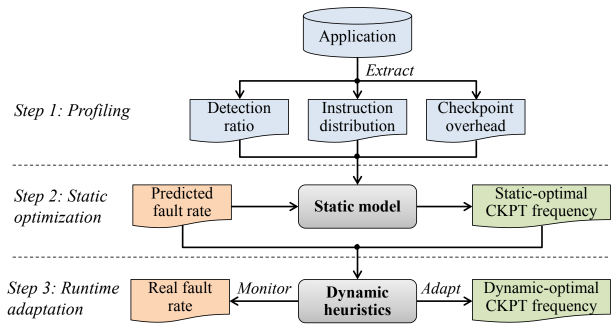


Fig. 2. Functional overview of the proposed framework

previous runtime approaches [30], [29], [27] that employ complex heuristics to predict system behavior, our runtime heuristics for fault monitoring and checkpoint adaptation are developed in a way that optimizes the associated hardware efficiency.

IV. Compile-time Overhead Modeling

In this section, we first introduce the proposed fault detection, checkpointing, and recovery framework, then perform a thorough mathematical analysis to identify the static-optimal checkpoint frequency under a given fault-rate. Our goal is to identify the optimal checkpoint interval capable of minimizing the overall overhead of fault resilience while at the same time delivering full fault coverage. Techniques [12], [13] that sacrifice fault coverage to reduce fault detection or recovery overhead are beyond the scope of this paper.

A. Detection, Checkpointing, and Recovery Process

We select redundant execution, one of the most widely adopted fault tolerance technique as our baseline approach. This allows the systems to detect and recover transient and intermittent faults that may occur in the execution pipeline. Regular storage structures such as caches, register files, and the main memory can be protected using error correction codes.

To efficiently detect faults, the proposed framework does not compare the result of each instruction. Instead, the fault detection is composed of three parts: (1) Prior to executing a store instruction, the store address and value are compared to ensure that results written to the cache are correct. (2) Upon reaching a checkpoint, the value of each register is compared to ensure the cleanness of the processor state. The cleanness of the cache state is ensured through adopting the cache-based checkpointing technique proposed in [16]. (3) To detect faults in the control flow, the branch outcomes of one thread will be buffered and delivered to the other thread for comparison.

Upon detecting a fault, either at a checkpoint or at a store/branch instruction, the recovery procedure is performed by restoring the clean processor and memory states committed at the last checkpoint. In contrast, if no fault has been detected upon reaching a checkpoint, the register values will be saved, and the cache state will be saved to memory using the technique proposed in [16].

B. Mathematical Model Construction

The goal of the mathematical model is to identify, for a given application, the optimal checkpoint interval that can minimize the overall fault tolerance overhead during the execution of an application.

During execution, the system consistently performs fault detection and checkpointing regardless of whether a fault is detected or not. While recovery is to be performed only when a fault is detected. The overhead *per checkpoint interval* can be expressed in the following equation:

$$O_{det} + O_{ckpt} + F * (O_{rec} + O_{det}) \tag{1}$$

with F denoting the failure probability and O_{det}, O_{ckpt}, and O_{rec} the overhead of detection, checkpointing, and recovery, respectively. The second O_{det} in the equation is to account for the fault detection performed during the re-execution process.

Assume that the size of the checkpoint interval is t. In the following parts of this section, we will identify the impact of t on failure probability F and on the three types of overhead O_{det}, O_{ckpt}, and O_{rec}. The optimal checkpoint interval that minimizes the overall overhead will be computed at the end.

1) Modeling failure probability: F, the probability of a fault being captured during a checkpoint interval t, is determined by two factors: the probability that a fault will be generated, and the probability that a generated fault will be detected.

Along with existing checkpoint models, our model assumes that the *failure rate* (i.e., the momentary probability of failure at one instruction) follows the Poisson distribution. In other words, each instruction has a constant rate of λ for producing an incorrect result. It has been shown that the probability that a fault will be generated at or before time t is $1 - e^{-\lambda t}$.

As discussed in Section III, a generated fault may end up being masked during execution in various ways. In the proposed framework, a generated fault will be detected only if it propagates to a store or a branch instruction or stays in a register at the checkpoint. The fault propagation process is highly influenced by the control and data flow of the application. We use A_{der} to denote the application-specific ratio of detected faults to all the generated faults. Therefore, the probability of a fault being captured during a checkpoint interval t can be represented using Equation (2a):

$$F = A_{der} * (1 - e^{-\lambda t}) \tag{2a}$$

$$F \approx A_{der} * \lambda t \tag{2b}$$

Clearly, the checkpoint interval t is expected to be far less than $1/\lambda$ which equals the mean-time-to-failure (MTTF) in the Poisson distribution (otherwise the probability of getting a fault between two consecutive checkpoints will be more than 50%). This implies that $\lambda t \ll 1$. Under first-order approximation, Equation (2a) can be rewritten as (2b).

2) Modeling overhead: The overall overhead of fault resilience is composed of three parts: fault detection overhead, checkpoint overhead, and recovery overhead.

Fault detection overhead is composed of the comparison operations performed for store instructions, for register values upon reaching a checkpoint, and for branch outcomes.

For each store instruction, two comparison operations need to be performed to check its value and address. The total number of comparisons performed for stores within a checkpoint interval t is determined by the distribution of store instructions in the application. Using A_{str} to denote the application-specific ratio

TABLE I. VARIABLES AND PARAMETERS USED IN THE MATHEMATICAL MODEL

Variables:		
t	–	size of checkpoint interval
$O_{overall}$	–	overall overhead
O_{ckpt}	–	checkpoint overhead per interval
O_{det}	–	detection overhead per interval
O_{rec}	–	recovery overhead per interval
Application and architecture parameters:		
T	–	length of the application
λ	–	failure rate per instruction
A_{der}	–	detected faults / all injected faults
A_{str}	–	store instructions / all instructions
A_{ctrl}	–	branch instructions / all instructions
A_{reg}	–	size of the Interger/FP register file
A_{save}	–	overhead for saving system states
A_{roll}	–	overhead for restoring system states

of store instructions, the total number of comparison operations performed for store instructions is $2A_{str} * t$.

Upon reaching a checkpoint, the proposed framework checks all the registers to ensure the cleanness of the processor state. The total number of comparison operations is determined by the size of the register file. This architecture-specific parameter is denoted as A_{reg}.

The cost for comparing branch outcomes is far less than the cost for comparing registers or stores since the outcome of a branch is a single bit while the values and addresses are usually 32-bit long. Using A_{ctrl} to denote the ratio of branch instructions, the total number of comparison operations performed for branch instructions is $A_{ctrl} * t/32$.

Summing up these three portions, the fault detection overhead per checkpoint interval can be expressed as:

$$O_{det} \propto 2A_{str} * t + A_{reg} + A_{ctrl} * t/32 \tag{3}$$

Checkpoint overhead is the cost to save clean processor states, including storing all the register values and the cache states to memory. This is an architecture-specific parameter that is represented as a constant A_{save} in our model. In other words, the overhead for creating a single checkpoint is independent of the size of the checkpoint interval, as shown below:

$$O_{ckpt} \propto A_{save} \tag{4}$$

Recovery overhead is the cost to recover the system from a faulty state. In the proposed framework this also includes two portions: the time it takes to restore the processor states to a clean state, and the time to re-perform the amount of computation rolled back. The first portion is an architecture-specific parameter that is represented as a constant A_{roll} in our model. The second portion is determined by the number of instructions to be re-executed. This is in turn determined by the exact position where the fault is detected, either at a store or branch instruction or at a checkpoint. We assume that on average, half of the checkpoint interval needs to be re-executed. Combining these two portions, we will have

$$O_{rec} \propto A_{roll} + t/2 \tag{5}$$

C. Static-Optimal Checkpoint Identification

Given t, the size of checkpoint interval, the total number of checkpoints that will be created is determined by the length of the program, T. Assuming equally spaced checkpoints, the entire program will have a total number of T/t checkpoint intervals. The overhead for each checkpoint interval satisfies Equation (1). Therefore, the overall overhead $O_{overall}$ is proportional to

$$\frac{T}{t} * [O_{det} + O_{ckpt} + F * (O_{rec} + O_{det})] \tag{6}$$

with the expression of F given in Equation (2b), and O_{det}, O_{ckpt}, O_{rec} in Equations (3)–(5).

The optimal checkpoint interval that minimizes $O_{overall}$ is the value of t for which $\partial O_{overall}/\partial t = 0$. It can be computed that t_{opt}, the optimal value of t satisfies

$$t_{opt} = \sqrt{\frac{2(A_{reg} + A_{save})}{A_{der}(1 + 4A_{str} + A_{ctrl}/16)}} * \sqrt{\frac{1}{\lambda}} \tag{7}$$

Equation (7) indicates that, for a given application, the optimal checkpoint interval t_{opt} is proportional to the square root of the failure rate. Meanwhile, t_{opt} is independent of the length of the application. Instead, it is influenced by three *application-specific* parameters, namely, the ratio of detected faults, the ratio of store instructions, and the ratio of branch instructions. It is also influenced by two *architecture-specific* parameters, namely, the size of the register file and the overhead for saving system states. For a given embedded system and a given application set, these parameters can be extracted at the compile-time. In other words, the first factor in Equation (7) can be pre-computed and represented as a constant for the runtime adaptation framework to use.

V. Runtime Adaption

The mathematical analysis performed in the last section confirms that the optimal checkpoint frequency is a function of the fault rate. If the fault rate is constant and can be predicted accurately, the static-optimal checkpoint interval t_{opt} would be sufficient for minimizing the overall overhead of fault detection and recovery.

Unfortunately, in most cases the fault rate is unpredictable. For example, transient faults are typically caused by alpha-particle strikes, cosmic rays, or radiation from radioactive atoms [31]. The exact fault rate will be influenced by the location of the system and the time (day or night) when the program is executed. Moreover, voltage and temperature fluctuations may accentuate various failing mechanisms during program execution. As a result, it is necessary to develop a runtime framework that utilizes the statically constructed model and tunes the checkpoint frequency adaptively to match the fault rate.

A. Fault Rate Monitoring

To model real system behavior, it is strictly assumed that the runtime framework neither has knowledge of the actual rate for generating a fault, nor has control over fault propagation. The only fault-related information that can be collected includes the total number of detected faults and the number of instructions executed between two consecutive faults.

Using a counter, the runtime system can easily monitor the reciprocal value of the fault rate, that is, the number of instructions executed between two consecutive faults. This *fault*

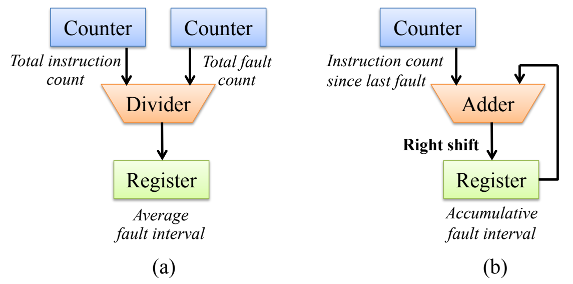


Fig. 3. Hardware for monitoring the fault interval

interval, denoted as $\Delta Fault$, equals $\frac{1}{A_{der}*\lambda}$ approximately. Due to the randomness nature of faults, however, each individual fault interval is more-or-less meaningless. It is therefore necessary to develop a heuristic to accumulate all the fault intervals and then adapt the checkpoint interval accordingly. It is furthermore necessary to pay utmost attention to the efficiency of the corresponding hardware implementation.

One possible implementation is shown in Figure 3(a), which computes the *average fault interval* through dividing the total number of executed instructions by the total number of detected faults. However, this implementation is quite inefficient due to its high hardware overhead. Since the application could be lengthy, large counters are needed for tracking the total number of instructions. A divider is also needed, which can be quite expensive and slow. More crucially, this implementation treats all faults equally: they all have equal impact on the value of the average fault interval regardless of their exact time stamp of detection.

Given these limitations in monitoring the average fault interval, we develop a more efficient heuristic that computes the *accumulative fault interval* (denoted as $\Delta Fault_{acc}$) using the following equation:

$$\Delta Fault_{acc} = \frac{\Delta Fault_{acc} + \Delta Fault_{ins}}{2} \tag{8}$$

The corresponding hardware implementation is shown in Figure 3(b). The counter value is incremented by one upon executing an instruction. Upon detecting a fault, the register value is updated while the counter value is reset to zero. This implementation is much more efficient than the one in Figure 3(a). Only a single counter is needed to keep track of the most recent fault interval (denoted as $\Delta Fault_{ins}$). No divider but only an adder is needed. The divide-by-2 operation in Equation (8) can be easily achieved through shifting the adder results 1-bit to the right. Furthermore, Equation (8) clearly confirms that for this implementation, a more recent fault has a larger impact on the value of $\Delta Fault_{acc}$. This implies that clustered fault behavior, if any, can be captured by this model.

B. Checkpoint Frequency Adaptation

With the accumulative fault interval monitored by the circuit in Figure 3(b), the runtime instant checkpoint interval can be determined accordingly. Yet one potential problem is the associated computation and hardware complexity, since Equation (7) requires an expensive and time-consuming square root operation to compute the checkpoint interval. To reduce such overhead, we develop a heuristic based on the well-known Babylonian method to quickly approximate the square root. For any number S and an initial seed r_0, this method iteratively computes a set of approximate values r_{i+1} using the following equation:

$$r_{i+1} = \frac{r_i + S/r_i}{2} \tag{9}$$

Clearly, if r_i is larger than $\sqrt{S}$, then S/r_i should be smaller than $\sqrt{S}$, and hence r_{i+1} should be closer to $\sqrt{S}$ than r_i.

While the standard Babylonian method can be repeated until a desired accuracy is reached, our runtime heuristic is constrained to a single-round approximation to reduce the associated overhead, implying that a high-quality initial seed is crucial. Considering the fact that hardware can efficiently implement divide-by-2 and multiply-by-2 through right shift and left shift operations, we decide to use first order approximation for the square root operation, as shown below:

$$\sqrt{S} \approx \begin{cases} S + 1/4 & \text{if } S \leq 1/2 \\ 1/2S + 1/2 & \text{if } 1/2 < S < 2 \\ 1/4S + 1 & \text{if } S \geq 2 \end{cases} \tag{10}$$

It turns out that for $S \leq 10$, Equation (10) is a good approximation for the square root operation. In light of this observation, we use the following heuristic to determine the runtime instant checkpoint interval t_{new}:

$$\frac{t_{new}}{t_{sta}} = \sqrt{\frac{\Delta Fault_{acc}}{\Delta Fault_{sta}}} \tag{11}$$

The static-optimal checkpoint interval t_{sta} and the static fault interval $\Delta Fault_{sta} = \frac{1}{A_{der}*\lambda}$ can be extracted at compile-time. These two values already embed the application- and architecture-specific parameters. The accumulative fault interval $\Delta Fault_{acc}$ will hardly be orders of magnitude larger or smaller than the static fault interval $\Delta Fault_{sta}$. As a result, Equation (10) will be a good approximation for $S = \frac{\Delta Fault_{acc}}{\Delta Fault_{sta}}$.

C. Overall Fault Propagation and Simulation Flow

Utilizing the aforementioned heuristics, the runtime instruction-level fault simulation can be performed in the flow shown in Algorithm 1. Each instruction is processed in four major steps: fault propagation, fault detection, recovery, and checkpoint.

Fault propagation is performed in lines 3–5. A set of poison bits ($Poison[\,]$) are used to keep track of fault propagation. If the instruction currently under execution has a destination register, the corresponding poison bit will be set to 1 if a new fault is generated for the current instruction or if its source register(s) have any fault.

Fault detection is performed in lines 6–18. The $Detect\_flag$ variable is used to track whether a fault is detected or not. Clearly, the detection process is composed of two parts. If the current instruction is a store or a branch instruction, its source operands will be compared, and $Detect\_flag$ will be set to 1 if any source register has a fault. The second part of the detection process is performed only upon reaching a selected checkpoint position. The entire register file is compared, and $Detect\_flag$ will be set to 1 if any register has a fault.

If any fault is detected in the above procedure, the simulator updates the values of $Fault_{acc}$ and then computes the next checkpoint interval using Equation (11). The t_{cnt} variable is used

TABLE II. FAULT INJECTION STATISTICS

	Program length	Total simulation runs			Total injected faults		
		λ=10^{-4}	λ=10^{-5}	λ=10^{-6}	λ=10^{-4}	λ=10^{-5}	λ=10^{-6}
adpcm	25,244,562	5	20	198	12,720	5,180	5,060
crc	533,385,975	5	5	10	267,723	26,694	5,383
g721	267,610,985	5	5	19	134,379	13,482	5,070
gsm	1,867,872,240	5	5	5	938,057	93,354	9,556
jpeg	15,513,321	5	33	323	7,689	5,121	5,036
mpeg2	1,251,298,497	5	5	5	627,903	62,746	6,185
rijndael	391,487,514	5	5	13	197,197	19,621	5,052

Algorithm 1 Fault propagation and simulation flow

```
1:  $t_{cnt} \Leftarrow 0$; $Poison$[0:Max-1] $\Leftarrow$ 0;
2:  while (execution not finished) do
3:      if (New fault generated) or (fault in any source register) then
4:          $Poison[dest] \Leftarrow 1$;
5:      end if
6:      $Detect\_flag \Leftarrow 0$;
7:      if (current instruction is store or branch) then
8:          Check source operands;
9:          if (fault in any source register) then
10:             $Detect\_flag \Leftarrow 1$;
11:         end if
12:     end if
13:     if (CKPT reached) then
14:         Check the entire register file;
15:         if (fault in any register) then
16:             $Detect\_flag \Leftarrow 1$;
17:         end if
18:     end if
19:     $t_{cnt}$++;
20:     if ($Detect\_flag$ = 1) then
21:         $(Fault_{acc} \Leftarrow Fault_{acc} + t_{cnt})/2$;
22:         $t_{cnt} \Leftarrow 0$;
23:         Update instant checkpoint interval based on $Fault_{acc}$;
24:         Clear all the 1's in $Poison$[ ];
25:         Roll back to last CKPT;
26:     else if (CKPT reached) then
27:         Create a new CKPT;
28:     end if
29: end while
```

to track the value of $Fault_{ins}$ in Equation (8). This variable is incremented by 1 upon executing an instruction (line 19), and reset to 0 upon detecting a fault (line 22). Subsequently, recovery is performed by restoring the last clean checkpoint, clearing all the 1's in poison bits, and then restarting the execution from the last checkpoint.

At the last step, if no fault is detected and the execution reaches a selected checkpoint position, checkpointing is performed in lines 26–28.

VI. EXPERIMENTAL EVALUATION

To implement and evaluate the proposed fault detection, checkpointing, and recovery framework, we extend the Simplescalar toolset [32] to implement fault injection and the simulation flow shown in Algorithm 1. Intensive experimental studies have been performed on Mediabench [33] and Mibench [34] programs.

A. Fault Injection

We assume that the failure rate follows the Poisson distribution. During execution, each instruction is allowed to produce an incorrect result at a constant rate of λ.

As each individual fault is completely random, it is necessary to generate a sufficient number of faults in order to capture the trend in overhead changes. Therefore, we set two thresholds to eliminate the impact of randomness: (1) run every single experiment configuration is at least 5 times, and (2) inject at least 5000 faults for each configuration tested. Unfortunately, injecting 5000+ faults requires a large number simulation runs to be performed *for every single experiment configuration*. The lower the fault rate, the larger the number of simulation runs an application needs to be tested. As a result, we are forced, by the slow speed of instruction-level simulation, to choose relatively higher failure rates. Specifically, in our experiments, the value of λ is set to 10^{-4}, 10^{-5}, and 10^{-6}.

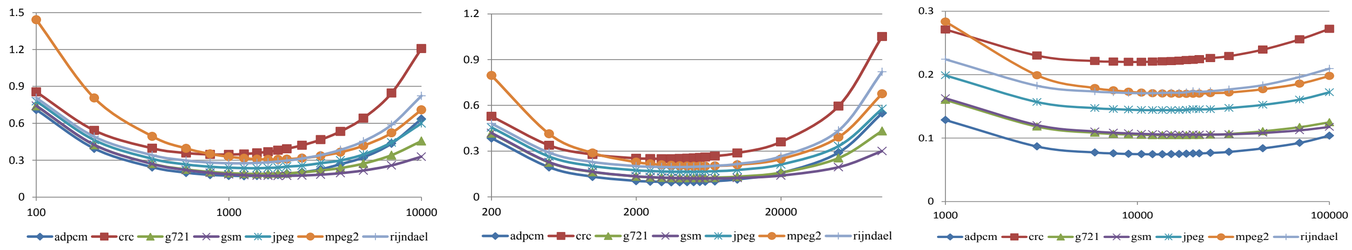
The statistics of our fault injection study is shown in Table II. Using N to denote the total number of instructions of a program, the number of simulation runs is computed as $max\{5, 5000/\lambda N\}$. The total number of faults injected during these simulation runs are also reported. It can be seen that for the smallest application *jpeg*, at the lowest fault rate of 10^{-6}, we performed as many as 323 runs in order to get the total number of injected faults close to 5000.

Using higher fault rates for evaluation, while seeming to be unrealistic at first sight, will not simplify the problem of fault resilience. Instead, we believe that techniques capable of solving higher fault rates will be effective for lower fault rates as well. We also think this study will be a meaningful approximation for future systems whose computation advances towards exascale and whose devices advance towards nanoscale.

B. Application Characteristics

Application characteristics are expected to have a noticeable impact on the overall efficiency of fault detection and recovery. This property is confirmed by the compile-time model: Equation (7) shows that the optimal checkpoint interval is determined by three application-specific parameters: the ratio of store instructions, the ratio of branch instructions, and the ratio of detected faults. These three parameters are reported in Table III. The first two are independent of the fault rate, while the last one is the average of all the random cases we have tested. It turns out that the value of fault rate has no observable impact on this parameter.

As can be seen, these three parameters vary across application significantly. For the ratio of store instructions and the ratio of branch instructions, $3.7X$ and $5.6X$ differences between the largest and the smallest can be observed, while a $2.4X$ difference can be observed for the ratio of detected faults.



(a) Fault rate λ=10^{-4} (b) Fault rate λ=10^{-5} (c) Fault rate λ=10^{-6}

Fig. 4. The change in the overall overhead as the checkpoint interval increases

C. Quality of the Compile-time Mathematical Model

To evaluate the compile-time mathematical model, we compare the predicted values obtained through this model against real data collected during fault simulation.

We have performed a large set of simulation studies to identify the real optimal checkpoint interval for each application. Three distinct fault rate configurations were tested: 10^{-4}, 10^{-5}, and 10^{-6}. Each configuration was tested with different checkpoint intervals, and the smallest steps for increasing the checkpoint interval are 200, 500, 1500 for these three configurations, respectively. The exact simulation runs performed at each case is reported in Table II.

The collected overhead values are normalized to the number of executed instructions in order to eliminate the impact of application length. These data, including both the fault detection and recovery overhead, are reported in Figure 4. The fault detection overhead is reported as all the comparison operations performed, either at store or branch instructions or at checkpoints. The recovery overhead is reported as the number of instructions re-executed due to a detected fault.

The curves in Figure 4 confirm the existence of optimal checkpoint intervals for all the tested applications. When the interval size is very small, the detection overhead dominates the overall overhead. Increasing the interval size will decrease the number of checkpoints needed, thus reducing the detection overhead and hence the overall overhead. As the interval size keeps increasing, the probability for a fault to occur during that interval is large enough such that the recovery overhead starts to dominate the overall overhead. Since then, increasing the interval size will cause the overall overhead to increase, as it not only elevates the probability of fault, but also requires larger amount of computation to be re-performed upon a fault.

As the fault rate decreases, the curves becomes flatter. This is due to the fact that the optimal checkpoint interval is reached when the decrease in fault detection overhead equals the increase in recovery overhead. However, when the fault rate is low, the recovery overhead becomes less significant and less sensitive to checkpoint interval. As a result, the curves of the overall overhead become flatter.

Using the same fault detection and recovery scheme, the various applications still display noticeable difference in optimal checkpoint intervals for the same fault rate. This clearly reflects the impact of the three application-specific parameters listed in Table III.

The real optimal checkpoint intervals, identified through this search process, are compared against the static-optimal checkpoint intervals computed using Equation (7). The three application-specific parameters are set to the values reported in Table III. The two architecture-specific parameters are set to the size of the register file in Simplescalar, that is, 32 for integer and 64 for floating-point applications. These predicted and real values of the optimal checkpoint interval are reported in Table IV. Note that the real values are constrained by the smallest steps for increasing the checkpoint interval, which are set to 200, 500, 1500 for the three fault rate configurations, respectively.

As can be seen, the predicted values are pretty close to the real values. Taking the impact of step size and the randomness in generating and detecting faults into consideration, we can conclude that these results confirm that the proposed compile-time mathematical model is effective in capturing application characteristics to predict optimal checkpoint intervals.

D. Effectiveness of Runtime Adaptation

To evaluate the effectiveness of the proposed runtime adaptation heuristics, we have performed studies on the following four schemes:

1) An optimal scheme that has knowledge of the exact rate of injected faults *a priori*. It uses the corresponding static-optimal checkpoint interval.

TABLE III. APPLICATION-SPECIFIC PARAMETERS

Benchmarks	Store %	Branch %	Detected fault %
adpcm	2.74%	29.85%	74.52%
crc	10.01%	20.01%	77.87%
g721	4.43%	23.00%	53.06%
gsm	4.84%	5.29%	34.39%
jpeg	6.50%	15.60%	63.56%
mpeg2	7.55%	11.01%	71.52%
rijndael	7.89%	5.77%	83.66%

TABLE IV. OPTIMAL CHECKPOINT INTERVALS, PREDICTED VS. REAL VALUES

	$\lambda = 10^{-4}$		$\lambda = 10^{-5}$		$\lambda = 10^{-6}$	
Benchmarks	Pred	Real	Pred	Real	Pred	Real
adpcm	1230	1200	3902	3500	12382	12000
crc	1093	1000	3407	3000	10663	10500
g721	1414	1400	4498	4500	14332	15000
gsm	1718	1800	5600	6000	18047	21000
jpeg	1247	1400	3990	4000	12696	13500
mpeg2	1651	1600	5233	5000	16552	18000
rijndael	1076	1200	3403	3500	10787	10500

2) A non-adaptation scheme that assumes a fault rate of 10^{-5} which is 10 times higher or lower than the actual fault rate. It uses the static-optimal checkpoint interval for 10^{-5}.
3) An adaptation scheme with exactly the same configuration and assumption as the non-adaptive scheme. It performs the exact square root computation of the fault interval to determine the next checkpoint interval.
4) An adaptation scheme with exactly the same configuration and assumption as the non-adaptive scheme. It approximates the square root values of the fault interval using Equation (10).

It is expected that the difference between schemes 2 and 3 will show the effectiveness of runtime adaptation, while the difference between schemes 3 and 4 will show the quality of our approximation, which is introduced to reduce the computational and hardware overhead.

For each scheme, two fault configurations in which the actual fault rate λ is respectively set to 10^{-4} and 10^{-6} (both differ from the predicted fault rate by 10X) are performed. Again, under each configuration, every application is tested for the number of times specified in Table II to reduce the impact of randomness. The overall fault detection and recovery overhead of these schemes is reported in Figure 5.

In Figure 5, the scheme without adaptation consistently performs worse than the other three schemes across all the configurations. This confirms that the performance of the static-optimal model is fully determined by how accurately the fault rate can be predicted. In contrast, the two adaptive schemes are able to achieve as good performance as the optimal one. This shows that even starting with an incorrect prediction of the fault rate, the runtime adaptation framework, with its ability to dynamically monitor the actual fault rate and adjust checkpoint frequency accordingly, outperforms the fixed, statically selected checkpoint interval. The difference between the accurate and the approximated schemes is negligible, implying that the light-weight square root approximation is very effective and accurate.

In Figure 5, the difference between the non-adaptive and adaptive schemes in the low fault rate (10^{-6}) case is not as significant as the the high fault rate (10^{-4}) case. This is because varying the checkpoint interval has a relatively smaller impact on the overall fault detection and recovery overhead when the fault rate is low. As a result, the non-adaptive scheme does not have a significant difference compared to the optimal case. This observation is consistent with the data reported in Figure 4, wherein the 10^{-6} curves are much flatter than the 10^{-4} curves.

VII. Conclusions

We have presented an integrated framework capable of minimizing the overall overhead of fault detection, checkpointing and error recovery. The framework integrates both compile-time analysis and runtime adaptation. A static model has been developed to mathematically characterize the application and architecture impact on checkpoint frequency. Under a predicted fault rate, this model is able to identify the optimal checkpoint frequency for an application. Meanwhile, to compensate for the lack of actual fault information at compile-time, a runtime adaptation framework is also developed. Two heuristics, developed with the goal of reducing the associated hardware overhead, allows the framework to dynamically monitor the clustered fault rate and adapt the checkpoint frequency accordingly.

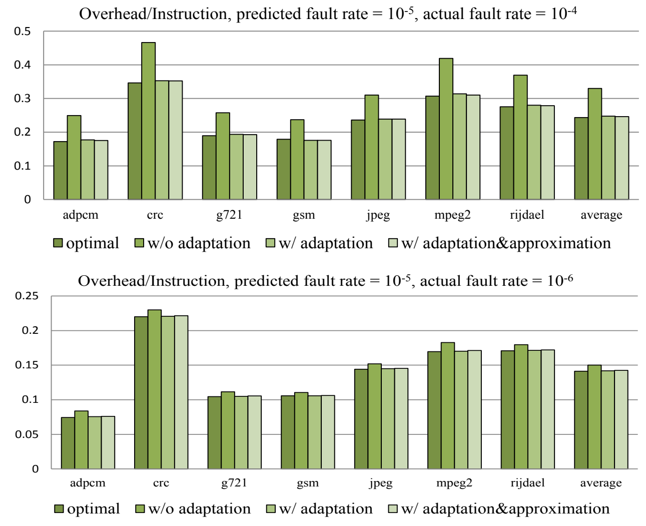


Fig. 5. Evaluation of dynamic heuristics

Intensive fault injection studies have been performed to evaluate the compile-time model and runtime heuristics. The experimental results show that the optimal checkpoint frequency identified by the static model is very close to the actual value (6% deviation on average). Moreover, when the actual fault rate is unknown *a priori*, the run-time adaptation scheme significantly outperforms the non-adaptation scheme, with its performance almost identical to the optimal one that knows the actual fault rate beforehand.

The integrated framework, as it is able to adaptively balance the overhead in fault detection and recovery, will help future embedded systems respond to the predicted increasing amounts and diverse types of hardware failures in the most efficient way.

References

[1] P. Shivakumar, S. W. Keckler, D. Burger, M. Kistler, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Intl. Conf. Dependable Syst. & Netw. (DSN)*, June 2002, pp. 389–398.

[2] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The impact of technology scaling on lifetime reliability," in *Intl. Conf. Dependable Syst. & Netw. (DSN)*, June 2004, pp. 177–186.

[3] "International Technology Roadmap for Semiconductors (ITRS) 2011 edition: Executive summary," 2011.

[4] R. Blish and N. Durrant, "Semiconductor device reliability failure models," International SEMATECH, Tech. Rep., May 2000.

[5] S. Borkar, T. Karnik, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *45th Design Autom. Conf. (DAC)*, June 2003, pp. 338–342.

[6] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, July 2003.

[7] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *29th Intl. Symp. Comput. Archit. (ISCA)*, May 2002, pp. 99–110.

[8] S. K. Reinhardt and S. S. Mukherjee, "Transient-fault detection via simultaneous multithreading," in *27th Intl. Symp. Comput. Archit. (ISCA)*, June 2000, pp. 25–36.

[9] M. A. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," *IEEE Micro*, vol. 23, no. 6, pp. 76–83, Nov. 2003.

[10] S. S. Mukherjee, S. K. Reinhardt, and J. S. Emer, "Incremental checkpointing in a multi-threaded architecture," in *US Patent Application*, Aug. 2003.

[11] T. Vijaykumar, I. Pomeranz, and K. Cheng, “Transient-fault recovery using simultaneous multithreading,” in *29th Intl. Symp. Comput. Archit. (ISCA)*, May 2002, pp. 87–98.

[12] M. A. Gomaa and T. N. Vijaykumar, “Opportunistic transient-fault detection,” in *32th Intl. Symp. Comput. Archit. (ISCA)*, 2005, pp. 172–183.

[13] V. K. Reddy, S. Parthasarathy, and E. Rotenberg, “Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance,” in *12th Intl. Conf. Archit. Support for Program. Lang. & OSs (ASPLOS)*, Mar. 2006, pp. 83–94.

[14] E. Rotenberg, “AR-SMT: a microarchitectural approach to fault tolerance in microprocessors,” in *29th Intl. Symp. Fault-Tolerant Computing (FTCS)*, June 1999, pp. 84–91.

[15] J. Sharkey, N. Abu-Ghazeleh, and D. Ponomarev, “Trades-offs in transient fault recovery schemes for redundant multithreaded processors,” in *13th Intl. Conf. High Perform. Computing (HiPC)*, December 2006, pp. 135–147.

[16] C. Yang and A. Orailoglu, “A light-weight cache-based fault detection and checkpointing scheme for MPSoCs enabling relaxed execution synchronization,” in *Intl. Conf. Compilers, Archit. & Synthesis for Embedded Syst. (CASES)*, Oct. 2008, pp. 11–20.

[17] S. K. Reinhardt, S. S. Mukherjee, and J. S. Emer, “Periodic checkpointing in a multi-threaded architecture,” in *US Patent Application*, Aug. 2003.

[18] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, “Fingerprinting: Bounding soft-error detection latency and bandwidth,” *IEEE Micro*, vol. 24, no. 6, pp. 22–29, Nov. 2004.

[19] H. Chen and C. Yang, “Boosting efficiency of fault detection and recovery throughapplication-specific comparison and checkpointing,” in *14th Conf. on Languages, compilers and tools for embedded systems(LCTES)*, June 2013, pp. 13–20.

[20] J. Wu, E. B. Fernandez, and D. Dai, “A first order approximation to the optimum checkpoint interva,” *Communications of the ACM*, vol. 17, pp. 530–531, September 1974.

[21] J. T. Daly, “A higher order estimate of the optimum checkpoint interval for restart dumps,” *Future Generation Computer Systems*, vol. 22, pp. 303–312, 2006.

[22] A. Duda, “The effects of checkpointing on program execution time,” *Information Processing Letters*, vol. 16, pp. 221–229, June 1983.

[23] Y. Ling, J. Mi, and X. Lin, “A variational calculus approach to optimal checkpoint placement,” *IEEE Trans. Computers*, vol. 50, pp. 699–707, July 2001.

[24] T. Ozaki, T. Dohi, H. Okamura, and N. Kaio, “Min-max checkpoint placement under incomplete failure information,” in *Intl. Conf. Dependable Syst. & Netw. (DSN)*, July 2004, pp. 721–730.

[25] Y. Liu, R. Nassar, C. B. Leangsuksun, N. Naksinehaboon, M. Paun, , and S. L. Scott, “An optimal checkpoint/restart model for a large scale high performance computing system,” in *22th Intl. Conf. Parallel & Distrib. Process. Symp. (IPDPS)*, April 2008, pp. 1–9.

[26] A. Ziv and J. Bruck, “An on-line algorithm for checkpoint placement,” *IEEE Trans. Computers*, vol. 46, pp. 976–985, 1997.

[27] A. Oliner and R. Sahoo, “Evaluating cooperative checkpointing for supercomputing systems,” in *20th Intl. Conf. Parallel & Distrib. Process. Symp. (IPDPS)*.

[28] Y. Zhang and K. Chakrabarty, “Energy-aware adaptive checkpointing in embedded real-time systems,” in *Design Autom. & Test in Europe (DATE)*, March 2003, pp. 918–923.

[29] M. Chtepen, F. Claeys, B. Dhoedt, F. D. Turck, P. Demeester, and P. Vanrolleghem, “Adaptive task checkpointing and replication: Toward efficient fault-tolerant grids,” *IEEE Trans. Parallel and Distributed Systems*, vol. 20, pp. 180–190, 2009.

[30] M. Chtepen, F. Claeys, B. Dhoedt, F. Turck, P. Demeester, and P. Vanrolleghem, “Adaptive checkpointing in dynamic grids for uncertain job durations,” in *31st Intl. Conf. on Information Technology Interfaces*, June 2009, pp. 585–590.

[31] T. Karnik, P. Hazucha, and J. Patel, “Characterization of soft errors caused by single event upsets in CMOS processes,” *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 2, pp. 128–143, 2004.

[32] T. Austin, E. Larson, and D. Ernst, “Simplescalar: an infrastructure for computer system modeling,” *IEEE Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.

[33] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, “Mediabench: A tool for evaluating and synthesizing multimedia and communications systems,” in *30th Intl. Symp. Microarchitecture (MICRO)*, Dec. 1997, pp. 330–335.

[34] J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “Mibench: A free, commercially representative embedded benchmark suite,” in *4th Workshop on Workload Characterization*, Dec. 2001, pp. 13–14.