Energy-efficient Adaptive Checkpoint Schemes for Computational Error Tolerant System

ABSTRACT

Denard scaling has end. Because of the leakage current, system energy saving via lower supply voltage is no longer feasible in traditional device, e.g. MOSFETs. Some recent proposed techniques should be able to mitigate this problem due to they can keep a high on/off ratio of drain currents in a much lower supply voltage. But the major constraint of the new technologies is they may suffer intermittent errors in high probability which finally leads system to crash. Computational error correction architectures aim to tolerate these errors and enable Denard scaling further extended. However, the energy saving abilities are restricted in current architectures due to they at most can tolerate only one error, which significantly limits the potential of further supply voltage turn down.

In this paper we exploit the error detection and correction properties of Redundant Residue Number System(RRNS). By carefully choose the RRNS base sets, our new architecture significantly improve the error tolerant ability which could detect all of the double residue errors and part of three or more residue errors. And we can safely remove the large size error correction table exists in RRNS error correction systems. Moreover, we design two energy-efficient adaptive checkpoint and rollback schemes: Extra Overhead Estimations and Error Interval Heuristics. Both of them are trying to make the best tradeoff between the EDP and reliability by automatically adjusting the long checkpoint intervals and incremental checkpoint intervals. On average, our new architecture with adaptive checkpoint scheme reduces 31.79% in energy consumption and 31.1% in EDP when comparing with the previous single error correction architecture. Provide more energy saving opportunities via lowing the Vdd.

1. INTRODUCTION

Dennard scaling was lost a decade ago due to ignoring the consideration of leakage current and threshold voltage limitation. The leakage current and threshold voltage does not scale but generating contributions to the power consumption per transistor. So power density soars with dimension decreasing and finally it hits the power wall.

Landauer [1] proposed the minimum energy dissipation is usually on the order of ‘kT’ for each of irreversible logic operation. The ‘k’ represents the Boltzmann’s constant and ‘T’ is absolute temperature. So at room temperature, the value of ‘kT’ is roughly equals to 4×10^{-21} joules. Complementary Metal-Oxide Semiconductor (CMOS) can never reach this minimum threshold. CMOS is the most popular MOSFET technology today and widely used in current commercial products such as memories, ASICs, microprocessors and other digital logic circuits. However, the simple CMOS charging circuit constrains the energy reduction at Landauer-Shannon limit [2], which is about 50X higher than the minimal value proposed by Landauer.

Some new devices, such as Tunnel FETs [3], could operate below the limits of MOSFET devices. MOSFETs are limited to ln(10) kT/q $\approx$ 60 mV/decade sub threshold slope. These new devices are known as millivolt switches. However, just like the signal strength of a cell phone would become weaker as it moves further from the transmitter tower, the error rate of millivolt switches increase with an expression like $P_e = exp(-E_{signal}/kT)$ [4]. So we must incorporate redundancy and error correction in logic or architecture design to keep the system error rate in bounds. In other words, if scaling continues with these devices go into production and some efficient error detection and/or correction schemes are proposed, technology should be able to go further to make the transistor signal energy between Landauer-Shannon limit and Landauer-Minimal.

1.1 Computational error detection and correction

Standard error correcting codes (ECC) [5] is the most common error detection and correction code and widely adopted in modern communication or storage system. However, ECC can’t detect the error from computational logic. For the upcoming new millivolt switch devices with higher error probability in low signal energy environment, we should design a computational error detectable architecture and minimize the possible error diffusion.

1.1.1 Triple Modular Redundancy(TMR)

The conventional approach for computational fault tolerance is called Triple Modular Redundancy(TMR) [6]. The basic idea of TMR is to duplicate the computational logic twice and then make a majority vote. If one of the computational logic is in error while remaining two are correct, this error could be simply corrected via the error-free majority voter. However, TMR needs more than 200% overhead in both area and power a single error detection and correction. Any energy savings from lowering Vdd would be eclipsed due to this overhead in correcting resultant errors.

1.1.2 Residue Checking

Residue checking is a popular scheme widely used in IBM commercial processors, such as z990, z10, z196, POWER6, POWER7 and etc. [7–10]. But it at least contains the following 4 constrains if used in low supply voltage configurations: (1) Higher checking overhead than RRNS with similar error detection capabilities. Most IBM processors use very small moduli(3, 9 or 15) so as to reduce the module complexity, but on the other hand these small numbers hurt the system error detection coverage(e.g. modulus 3, 1/3 errors are undetectable). Because the transient fault ratio should be much higher in near-threshold voltage, the system MTBF significantly goes down if adopting the residue checking method with mini moduli. Increasing the modulus to a large number(e.g. 251*509) is necessary to reach the reasonable error detection capability. However the side-effect of using a large modulus is increasing the module logic complexity in both area and energy consumption, leading the overhead of a single residue checking is much worse than a single RRNS verification. (2) Residue Checking is necessary after every arithmetic operation. If we don't perform residue check after a given arithmetic operation, then an error in that operation goes undetected forever. However, with RRNS, you can perform the RRNS checking at any time because the verification information remain exist in redundant residues. 3) The input data should be verified (e.g. ECC checking) before arithmetic computation. If one of the inputs is in error, this residue checking circuit is unable to detect it. The residue checking can only detect the error during the ALU operations. This input checking is unnecessary in RRNS system. 4) Unable to optimize the ALU because the data are in binary format. If the data are in RRNS format, we can use index-sum algorithm [11] to simplify the multiplication in ALU. However, the data in residue checking are in binary format. Binary data are not suitable to use the index-sum method because the related LUT size would be pretty huge.
One advantage of residue checking is it may get some latency benefits from a single check. The IBM residue checking operation at least takes 3 cycles while the cost of RRNS check is at most 8 cycles. But in the low energy system design, the latency isn't the first consideration. Moreover, both residue checking and RRNS checking latencies could be hided because none of them is in the critical path.

1.1.3 RNS and RRNS

The Residue Number System(RNS) has been used as an alternative to the binary number system chiefly to speed up computation [12, 13]. This increased efficiency comes from the fact that a large integer can be represented using a set of smaller integers, with arithmetic operations permissible on the set in parallel. We present some of the properties of RNS below.

Let $B = \{m_i \in \mathbb{N}\ for\ i = 1, 2, 3, ..., n\}$ be a set of n co-prime natural numbers, which we shall refer to as bases or moduli. $M = \prod_{i=1}^{n} m_i$ defines the range of natural numbers that can be injectively represented by RNS that is defined by the set of bases B. Specifically, for x such that $x \in \mathbb{N}\ and\ x < M$, then, $x \equiv (|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, ..., |x|_{m_n})$, where $|x|_m = x\ mod\ m$. Each term in this n-tuple is referred to as a residue.

We also note that addition, subtraction and multiplication are closed under RNS. This is because of the following observation: given $x, y \in \mathbb{N}\ and\ x, y < M$, we have $|x\ op\ y|_m = ||x|_m\ op\ |y|_m|_m$, where *op* is any add/subtract/multiply operation.

To augment RNS with fault tolerance, r redundant bases are introduced. The set of moduli now contains n non-redundant and r redundant moduli: $B = \{m_i \in \mathbb{N}\ for\ i = 1, 2, 3, ..., n, n+1, ..., n+r\}$. The reason these extra bases are redundant is because any natural number smaller than $M\ (= \prod_{i=1}^{n} m_i)$ can still be represented uniquely by its n non-redundant residues. Intuitively, the r redundant residues form a sort of *error code* because all residues are transformed in an identical manner under arithmetic operations. For x such that $x \in \mathbb{N}$, $x < M$, then, $x \equiv (|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, ..., |x|_{m_n}, |x|_{m_{n+1}}, ..., |x|_{m_{n+r}})$ contains n non-redundant residues as well as r redundant residues. For convenience, we further define $M_R = \prod_{i=n+1}^{n+r} m_i$.

Upon applying arithmetic transformations to an RRNS number, any error that occurs in one of the residues is contained within that residue and does not propagate to other residues. When required, such an error can be corrected with the help of the remaining residues. Specifically, an RRNS system with (n, r) = (4, 2), a single errant residue can be corrected, or, two errant residues can be detected. Table 1 provides a simple example, Deng etc. [11] outlines necessary algorithms to do so. Research by Watson and Hastings [14–16] lays the foundation for the underlying theoretical framework that is used and extended in our work. Their work also details algorithms to handle RRNS scaling and fractional multiplication. They used (199, 233, 194, 239, 251, 509) as the (4, 2)-RRNS system, providing a range $M = 199 \times 233 \times 194 \times 239 \in (2^{31}, 2^{32})$. In Section 2.1, we discuss the methodology and implications of choosing a different set of RRNS bases for the purposes of trading range with overhead.

Not only does a residue number system achieve a higher efficiency due to enhanced bit-level parallelism (also, no carries required for addition), but also that introducing 50% of overhead is sufficient to provide resiliency. As the granularity of an error is that of an entire residue, RRNS is capable of potentially correcting multi-bit errors as well, for *free*.

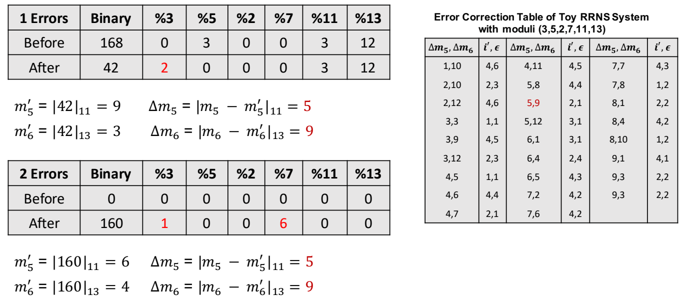
We design a computer based on these properties.

Table 1: A (4, 2)-RRNS example with the simplified base set (3, 5, 2, 7, 11, 13). Range is 210, with 11 and 13 being the redundant bases.

Decimal	mod 3	mod 5	mod 2	mod 7	mod 11	mod 13
13	1	3	1	6	2	0
14	2	4	0	0	3	1
13+14=27	(1+2)mod 3=0	2	1	6	5	1
	All columns function independently of one another.					
	An error in any one of these columns (residues) can be corrected by the remaining columns.					

1.2 CREEPY and limitations

Deng etc. [11, 17, 18] proposed a Single Error Correction(SEC) RRNS architecture with 4 non-redundant residues and 2 redundant residues, which is known as CREEPY or (4,2)-RRNS SEC. Actually (4,2)-RRNS configuration is able to correct single error residue or detect any of 2 error residues. However, the property of RRNS doesn't allow it to support both capabilities in a same system. In other words, only one of them can be selected(CREEPY is SEC). The reason is that the delta value pairs from the output of RRNS checking algorithm may be overlapped in SEC and DED mechanism. Figure 1 gives an detailed example to explain this phenomenon. We use a simple toy (4,2)-RRNS base set (3,5,2,7,11,13) for easily computing. And the range of this toy set equals to $3 \times 5 \times 2 \times 7 = 210$, which is the product of all the non-redundant moduli.In Case 1, the original correct RRNS value is (0,3,0,0,3,12), and then the system gets one residue in error(the 1_{st} residue change from 0 to 2). If a RRNS check is inserted after this error, by using the Chinese Remainder Theorem [19] or Base Extension Algorithm [15, 16], the system gets a value pair $m_5{}'$ and $m_6{}'$. By making a modulus subtracting with the output of related redundant subcore, finally the system gets the delta value pair $(\Delta m_5, \Delta m_6)$. This delta value pair is an important output which directly relates to the RRNS residue error information. If both delta values equal to zero, it means that no error in this RRNS data; If one of them doesn't equal to zero, the corresponding redundant core should be in error. The correction of this case is straightforward, we can use the output of Chinese Remainder Theorem or Base Extension Algorithm to replace the error redundant subcore. If both delta values don't equals to zero, it means the error is in non-redundant subcore and we have to check the error correction table. Delta value pair (5,9) is used as the input to search the table, and we found the entry with a related output (2,1). The output (2,1) means the system should add a correction factor ̀1̀to the second non-redundant subcore to make the correction. In the one error example, because all the computation in subcore 2 would automatically module 3, so this one residue error could be fixed by adding this correction factor. This mechanism works well if it only supports SEC. But if the system is SECDED, from the 2 error example of Figure 1, we can finally get a same delta value pair $(\Delta m_5, \Delta m_6)$. If checking the error correction table and adding the correction factor in this double error case, the output will be wrong. It should rollback to the previous valid checkpoint and re-execute. In other words, the system may be unable to make the right decision for correction or rollback if it supports SECDEC. So only one of the capability, SEC or DEC, could be used in (4,2)-RRNS system.



1 Errors	Binary	%3	%5	%2	%7	%11	%13
Before	168	0	3	0	0	3	12
After	42	2	0	0	0	3	12

$m_5' = |42|_{11} = 9$ $\quad \Delta m_5 = |m_5 - m_5'|_{11} = 5$

$m_6' = |42|_{13} = 3$ $\quad \Delta m_6 = |m_6 - m_6'|_{13} = 9$

2 Errors	Binary	%3	%5	%2	%7	%11	%13
Before	0	0	0	0	0	0	0
After	160	1	0	0	6	0	0

$m_5' = |160|_{11} = 6$ $\quad \Delta m_5 = |m_5 - m_5'|_{11} = 5$

$m_6' = |160|_{13} = 4$ $\quad \Delta m_6 = |m_6 - m_6'|_{13} = 9$

Error Correction Table of Toy RRNS System with moduli (3,5,2,7,11,13)

$\Delta m_5, \Delta m_6$	i', ϵ	$\Delta m_5, \Delta m_6$	i', ϵ	$\Delta m_5, \Delta m_6$	i', ϵ
1,10	4,6	4,11	4,5	7,7	4,3
2,10	2,3	5,8	4,4	7,8	1,2
2,12	4,6	5,9	2,1	8,1	2,2
3,3	1,1	5,12	3,1	8,4	4,2
3,9	4,5	6,1	3,1	8,10	1,2
3,12	2,3	6,4	2,4	9,1	4,1
4,5	1,1	6,5	4,3	9,3	2,2
4,6	4,4	7,2	4,2	9,3	2,2
4,7	2,1	7,6	4,2		

Figure 1: An Example of SECDED conflict in a same (4,2)-RRNS system

In CREEPY, two or more residues in error are defined as failure. But if (4,2)-RRNS DED is used, the failure threshold could be upgraded to three or more residues in error. This gives the potential for the signal energy further turned down. Another possible option for error detection is to cut down one redundant residue and make it as (4,1)-RRNS Single Error Detection(SED). This may reduce each of computation and RRNS checking overhead.

From the simulation result in Section 5, (Energy Delay Product)EDP can further reduce 30-40% when comparing with CREEPY if designing an efficient checkpoint and rollback mechanism with double error detection infrastructure. Moreover, CREEPY needs a large size error correction table which stored in ROM, significantly increasing the hardware and area overhead.

1.3 Contributions

1. Deisgn (4,1)-RRNS and (4,2)-RRNS checkpoint systems with reasonable base sets and supported incremental checkpoint mechanism.

2. By using the RRNS checkpoint, we can remove a ROM unit which stored the large size error correction table.

3. Proposed 2 adaptive checkpoint interval adjustment methods and finally reducing 30-40% of energy and EDP compared with CREEPY.

4. Improve the feasibility of RRNS system. In (4,2)-RRNS DED configuration, the related error model could tolerate at most two errors between two checks. But in CREEPY, it only allows one error between two checks.

2. RRNS CHECKPOINT SYSTEM ARCHITECTURE

As discussed in Section 1.2, we have 2 possible solutions to further reduce the energy consumption when comparing with CREEPY: (4,1)-RRNS Single Error Detection (SED) and (4,2)-RRNS Double Error Detection (DED). Because lack of error correction ability in these two configurations, an efficient error detection and rollback mechanism is necessary for recovery from error or failure. More details about the adaptive checkpoint and rollback schemes will be discussed in Section 3.

Figure 2 shows the architecture overview of the two possible RRNS Checkpoint and Rollback Systems. Each RRNS core includes 4 non-redundant subcores and 1 or 2 redundant subcores. The Instruction Register is shared by all the subcores and protected by ECC. Similar to CREEPY architecture, ICache is shared by all the subcore while Dcache is distributed in each of the subcore(each subcore only store one residue of the related data). And the Memory could be protected via efficient ECC and assuming error free in this paper.

The architecture of (4,2)-RRNS DED configuration is similar to CREEPY. The biggest difference is no large size error correction table in these checkpoint and rollback system. The Residue Interactive Unit(RIU) consists of RRNS-Binary Conversion Unit and Consistency Checking Unit. The number of Consistency Checking Unit depends on the amount of redundant subcore. (4,2)-RRNS DED needs two Consistency Checking units while (4,2)-RRNS DED only needs one. Because of less dynamic energy consumption in each of the computation and in each of consistency checking, (4,1)-RRNS SED may have the potential to get energy benefit even though it may loss some reliability. On average, one third of error in (4,1)-RRNS SED system is undetectable, and this undetectable error is also defined as one kind of failure. So the MTBF of (4,1)-RRNS SED system should be lower in a same signal energy level. In other words, we may need higher signal energy to keep reliability. For the (4,2)-RRNS DED, because it can detect all the 1-residue or 2-residue errors and majority of multiple-errors if the RRNS base set is carefully selected, the MTBF should be higher at a same signal energy level. So signal energy of (4,2)-RRNS DED system could be further turned down and probably to save system energy if an efficient checkpoint mechanism is used.

2.1 Bases Selections for Both Systems

The base sets used in the RRNS core depend on the number of redundant residues and the error detection/correction capability. E.g, in (4,2)-RRNS, the base set candidates should be different for single error correction(SEC) and double error detection(DEC). The SEC base set options are discussed in CREEPY [11]. Because the RRNS checkpoint mechanism is based on the theory of DED, we list the necessary and sufficient conditions for (4,2)-RRNS double error detection here:

1. Each pair of bases m_i,m_j must be relatively prime.

2. All the products of the 2 moduli taken 2-(2-j) at a time are greater than the maximum product of j N-R moduli, where j =1 or 2.

3. $|m_1 m_2 - m_3 m_4| = 1$; also known as the K, K-1 property. (For RRNS fractional multiplication.)

4. mi $\in$ {x | x is either prime (p), a power of prime (p^m), or a power of 2 (2^m) }. Recall that the relative order of preference is p $> p^m > 2^m$, and that smaller bases result in smaller ROMs. (For index sum multiplication.)

The proof of these conditions are available in [15]. Table 2 are the base set candidates of (4,2)-RRNS DED system. The total number of *Core Bit*, *Range* and *Base Format* are there factors we should consider. The *Core Bit* directly relate to the area, hardware and energy overhead. For the *Range*, larger is better. The *Base Format* is related to the index-sum multiply algorithm [11] and the preference order is p $> p^m > 2^m$. The base set proposed by Waston [15] and the range of 32-bit binary are added in the table for comparision. If preferring low energy, less hardware and no large numbers in the applications, the base set (113,239,128,211,241,251) would be the best choice. If range is an important factor, (139,349,128,379,509,503) may be a good option. If you want the range better than 32-bit binary, it has to choose (211,421,256,347,509,503). Actually, this base set is only 1 bit more than (139,349,128,379,509,503) in total *Core Bit* while it gets more than 2 times of range increment.

Because (4,1)-RRNS system only includes one redundant residue, which means that it can detect a single residue in error. The necessary and sufficient conditions for (4,1)-RRNS single error detection actually is a subset of (4,2)-RRNS double error detection. Simply changing condition (2) of (4,2)-RRNS DED to “The redundant modulus must be greater than the largest non-redundant residue“, we can get the limitations for (4,1)-RRNS. Due to the space limitation, we don’t list the (4,1)-RRNS base set candidates in the paper.

2.2 RRNS Checkpoint and Rollback System Overview

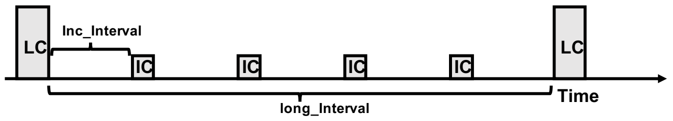
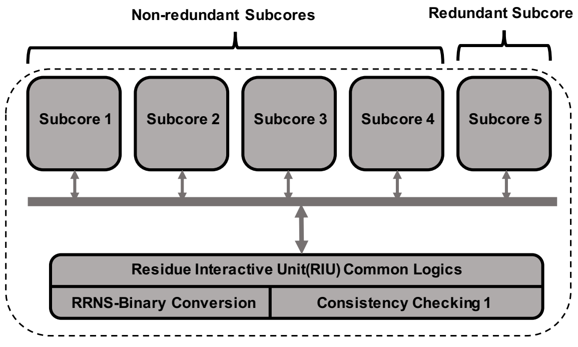
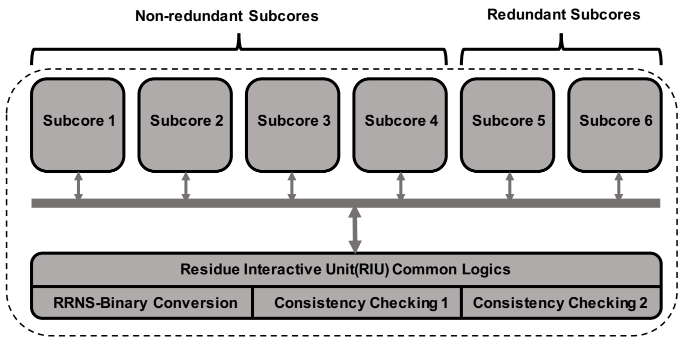


Figure 3: RRNS Checkpoint Mechanism Flowchart



(a) (4,1)-RRNS Single Error Detection(SED) Core



(b) (4,2)-RRNS Double Error Detection(DED) Core

Figure 2: RRNS Checkpoint and Rollback Core Overview

Table 2: Possible (4,2)-RRNS Base Sets

Subcore Bits	Core Bit	Range	Possible Base Sets	Base Format
7, 8, 7, 8, 8, 8	46	467921792	(97,223,128,169,241,251)	$(p, p, 2^7, 13^2, p, p)$
7, 8, 7, 8, 8, 8	46	729405056	**(113,239,128,211,241,251)**	$(p, p, 2^7, p, p, p)$
8, 8, 7, 8, 8, 8	47	635871872	(151,167,128,197,241,251)	$(p, p, 2^7, p, p, p)$
7, 8, 8, 7, 9, 9	48	430002432	(89,233,256,81,509,503)	$(p, p, 2^8, 3^4, p, p)$
7, 9, 7, 8, 9, 9	49	951568256	(109,283,128,241,509,503)	$(p, p, 2^7, p, p, p)$
7, 9, 7, 8, 9, 9	49	1032240512	(89,361,128,251,509,503)	$(p, 19^2, 2^7, p, p, p)$
8, 8, 8, 8, 8, 9	49	2149852322	(199,233,194,239,251,509)	$Waston$
9, 7, 7, 9, 9, 9	50	1082179712	(491,67,128,257,509,503)	$(p, p, 2^7, p, p, p)$
7, 9, 7, 9, 9, 9	50	1406512512	(81,463,128,293,509,503)	$(3^4, p, 2^7, p, p, p)$
7, 9, 8, 8, 9, 9	50	1230080256	(81,433,256,137,509,503)	$(3^4, p, 2^8, p, p, p)$
8, 9, 7, 9, 9, 9	51	2353365632	**(139,349,128,379,509,503)**	$(p, p, 2^7, p, p, p)$
		4294967296		32-bit binary
8, 9, 8, 9, 9, 9	52	7891035392	**(211,421,256,347,509,503)**	$(p, p, 2^8, p, p, p)$
9, 9, 8, 9, 9, 9	53	7710332672	(277,317,256,343,509,503)	$(p, p, 2^8, p^m, p, p)$

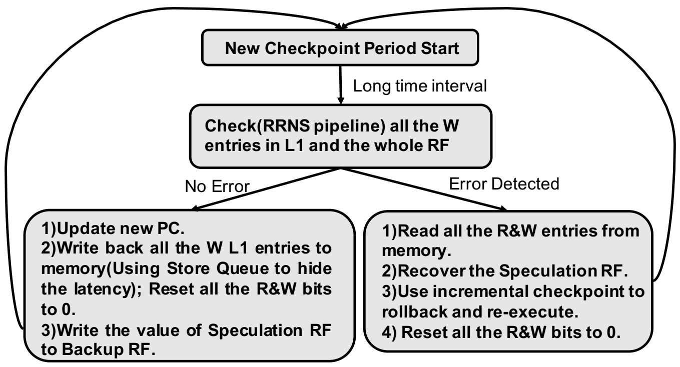

New Checkpoint Period Start
Long time interval
Check(RRNS pipeline) all the W entries in L1 and the whole RF
No Error
Error Detected
1)Update new PC.
2)Write back all the W L1 entries to memory(Using Store Queue to hide the latency); Reset all the R&W bits to 0.
3)Write the value of Speculation RF to Backup RF.
1)Read all the R&W entries from memory.
2)Recover the Speculation RF.
3)Use incremental checkpoint to rollback and re-execute.
4) Reset all the R&W bits to 0.

Figure 4: RRNS Checkpoint Mechanism Overview With Incremental Checkpoints

Figure 3 gives the overview about how RRNS checkpoint mechanism works with incremental checkpoint [20–22]. The Long_Interval could be fixed or variable, and it could be divided into several short time interval(Inc_Interval) and using incremental checkpoint to reduce the re-execution overhead. The overhead reduction depends on the error generation time and the number of incremental checkpoints in a long interval. More discussions about the incremental checkpoint are available in Section 3.2. After each of the Long_Interval, the system inserts a Long Check(LC) to verify the correctness and update the system status, the details of LC operations are shown in Figure 4. In the current error model, we assume the memory is error free due to many protection techniques**??** are available. At the beginning of the LC, the system checks the whole Speculation RF in pipeline way [11, 17] and all the modified entries of cache(W entries). It should be noted that we don't need to check the R cache entries. Because the load value and its related results would be immediately spread to Speculation RF or write back to cache. These values have to be verified during LC. If no error is detected, we can simply update the PC, cache and Register File. But if one or more errors is detected, the system should recover to the previous verification status and then re-execution.

The cache eviction behaviors become tricky in checkpoint mechanism and we can classify them into 3 cases:

1. R==0 and W==0: The cache block can be directly evicted without any RRNS checking. Errors in this line won't affect the correctness pf result because nobody read it and it won't write back to the memory.

2. R==1 and W==0: The cache block can be directly evicted without RRNS checking. Because these load errors would spread and be detected in RF or/and in 'W' cache entries. Moreover, sometime the error may happen after the last usage. So we can directly ignore these errors because they won't affect the result.

3. W == 1: Need a RRNS check before eviction. If one or more errors is detected, rollback to the previous correct status; If no error is found, this line

Table 3: equation terminologies

Name	Explaination
All_full	Overall full checkpoint verification and update overhead between the last 2 errors
All_inc	Overall inc checkpoint creation overhead during the same time-slot
$E(X)$	Expected value of error cycle in the last long checkpoint interval
$Long\_interval$	The time interval between 2 full checks
num_inc	Number of incremental checkpoints in a single long checkpoint interval
ave_long	average overhead of long checkpoint interval between the last 2 errors

should be stored in a writeback buffer (It can't be written back to main memory immediately because this may change the current correct state of memory). Committing the writeback buffer to memory only after the whole checkpoint verification process completes.

3. CHECKPOINT AND ROLLBACK SCHEMES

3.1 Long Storage Entry Verification Interval

RRNS Check all the store values(extra queue) and destination register values in this time interval. This interval should be as long as possible while keeping good reliability.

3.2 Incremental Checkpoint scheme

Why need incremental checkpoint????? Hard to find out the best static check intervals. Even the suboptimal interval configurations depends on the actual workload. So adaptive methods are necessary.

3.3 Extra Overhead Estimation

The long checkpoint interval is a critical factor for optimizing system energy and performance. In order to get the efficient tradeoff, It's necessary to figure out a reasonable checkpoint interval determination scheme. The scheme proposed in this section is called Extra Overhead Estimation(EOE). The basic idea of EOE is trying to estimate the next interval overhead based on the history information. Then making the decision of increasing, decreasing or keeping the old long check interval value.

After the checkpoint verification procedure and no error detected, the system computes the following 3 fomulas:

$$All\_full + All\_inc + \frac{\lfloor (1 - \frac{E(X)}{Long\_Interval}) \times num\_inc \rfloor + 1}{num\_inc} \times ave\_long \tag{1}$$

$$All\_full \times 0.5 + All\_inc + \frac{\lfloor (1 - \frac{E(X)}{Long\_Interval}) \times 2 \times num\_inc \rfloor + 1}{num\_inc \times 2} \times 2 \times ave\_long \tag{2}$$

$$All\_full \times 2 + All\_inc + \frac{\lfloor (1 - \frac{E(X)}{Long\_Interval}) \times 0.5 \times num\_inc \rfloor + 1}{num\_inc \times 0.5} \times 0.5 \times ave\_long \tag{3}$$

The Formula (1) computes the total overhead between last 2 errors; The Formula (2) roughly estimates the total overhead if doubling the long time interval during the last 2 errors; Formula (3) roughly estimates the total overhead if the halving the long time interval during the last 2 errors. The terminologies of these 3 formulas are explained in Table 3. In order to compute the results of these formulas, we have to figure out the expected value of error time in the last long check interval. The details about how to compute the expected value E(X) are available in APPENDIX .1. If Formula(1) gets the smallest value, the system keeps the long check interval value unchanged; If Formula(2) gets the smallest value, the system double the long check interval value; Similarly, If Formula(3) gets the smallest value, the system halves the long check interval value.

3.4 Error Interval Heuristics

Error Interval Heuristics(EIH) mechanism is an empirical adaptive method. It's based on an assumption that the time intervals between 2 sequence errors should be similar in most cases. Based on this assumption, the probability of the error is detected in the first half time is lower, but the chance would increase as the time past. So we can insert less RRNS checks in the first half time and gradually increasing the check frequency in exponential way. Figure 5a describes how the EIH works before the error is detected. In this example, we assume the last two errors interval is 200,000 cycles. In the 1^{st} iteration, long check interval is set to 200,000/2=100,000 cycles with no incremental checkpoint. Less check frequency is benefit from reducing system energy and performance loss. In the 2^{nd} iteration, system halves the old long check interval and insert one incremental checkpoint. Similarly, for the future iterations, we halve the last long check interval and double the number of incremental checkpoints. Minimal values for long interval and incremental interval are set because the overhead may exponentially increase if higher frequency long checks and/or incremental checks are used. In this example, we set the minimal long check interval to 30,000 cycles while the minimal incremental checkpoint interval is 10,000 cycles. So in iteration 3, the long interval is set to 30,000 instead of 25,000. For the iteration 4, the system keeps the old intervals due to both of them reach the limitations. In Figure 5a would keep this checkpoint interval configuration until the 1^{st} error is detected. The system only performs RRNS checks at the end of current long interval (The L blocks in Figure 5), so it should always has a delay between error generation and error detection.

Figure 5b gives the details about how to handle the 1^{st} error. Once the 1^{st} error is detected at the end of iteration 3, we should rollback to the last verified full system status (At the end of the2^{nd} iteration), including PC, Register File and all the W cache blocks. Then checking the incremental checkpoints in sequential order. The 1^{st} incremental checkpoint should be correct should write all the record data to the system. Similar checking the next incremental checkpoint until the error is detected. In this example the 2^{nd} incremental checkpoint is error and the system should re-execute from the status right after the 1^{st} incremental checkpoint. Set the new error interval value equals to summation cycles of these 3 iterations(100,000+50,000+30,000+checkpoint creating and RRNS verification time) and start a new execution process.

We onIy discussion one error detection above. If two or more errors are found in a same long interval, this may imply high error probability and the system needs higher frequency check. So in this case, the error interval is set to the value of last long interval divided by number of errors. Then start the new execution.

4. EVALUATION METHODOLOGY

To measure the performance-energy-reliability trade-off of different RRNS cores, we augment a stochastic fault injection mechanism into a cycle-accurate in-order trace-based simulator. We abstract the notion of using next-generation devices operating at low signal energies (E_s) and the resulting interaction with the kT noise floor into P_e, the probability of an error occurring in a transistor state in any given cycle. E_s, provided as one of the input to the simulation, is a measure of the signal energy at the input of a transistor; P_e is the probability of a fault occurring at the output of a transistor in any given cycle. The relationship of E_s and P_e can be defined by the following equation: $P_e = exp(\frac{-E_s}{kT})$. Similar to the CREEPY simulator [11, 17], these inputs are vectors as they denote the signal energies and error probabilities for each voltage domain, however, for the simple explanation purposes, we present them as scalars in the remainder of this section. Also input to the simulator is the check insertion strategy. The RRNS check could be placed in static or adaptive methods. Because we are evaluating a very different number system, we simulated an unpipelined microarchitecture with no branch prediction and a 2-level memory hierarchy (LLC-DRAM, with latencies of 12 cycles and 100 cycles for LLC hit and miss respectively) to maintain our primary focus in this paper. Adding more memory features to our design has been left as future work.

We first introduce a series of error events and their probabilities.

P_e Probability of an error occurring in a transistor state in any given cycle. This is provided as an input to the simulation, as just discussed.

P_{add} Probability of at least a single error in an adder (each sub-core has an adder in ALU). If there are N_{add} transistors in an adder, the probability of each of these transistors being free of error is $(1-P_e)^{N_{add}}$. Therefore, P_{add}=1-$(1 - P_e)^{N_{add}}$. Similarly, P_{sub} and P_{mul} are calculated. For multi-cycle operations, this definition holds as long as the state of each transistor is used exactly once for the operation. This is true for the said operators. Note that this is a conservative (pessimistic) estimate in our evaluation because we ignore any error masking that may potentially occur.

P_{R_i} Probability of at least 1 error being present in a slice (sub-core/residue) of register R_i since its last write. To compute this, we devise a $StateTable$, the i^{th} entry of which holds the tuple (P, $cycle$), where, P is the probability of R_i having atleast 1 error being present in the corresponding residue upon its most recent update at cycle $cycle$. This $StateTable$ is updated for each register write.

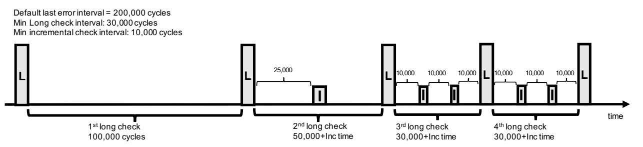
For example, consider the register R_0. 1) At cycle 0, the default value of R_0 tuple is (P=0, cycle=0). 2) At cycle 10, assume that we have an ADD instruction: ADD R0, R1, R2, and that it is the first instruction writing to R_0. We then update the tuple value to (Error_Probability_ADD, 10). It is necessary to update the P value here because the error probability of this ADD instruction should be taken into account. P value would then be set back to 0 once an RRNS check is inserted for that register and no error is detected, and then set the current system cycle value to the cycle field. This way, the P field in the $StateTable$ always reflects the probability of that register of having at least 1 error being present in one of its residues, given its most recent update at the cycle field.

Assuming an SRAM implementation of 8-bit wide R_i, the number of transistors is $8 \times 6 = 48$. The probability of R_i being error free is subject to two probabilities: (1) probability of an error-free write, ($P_1 = 1 - StateTable[R_i].P$) and, (2) probability of no error creeping into it since its last write ($P_2 = (1 - P'_e)^{48(c - StateTable[R_i].cycle)}$), where, c is the current cycle and P'_e is the probability of an error occurring in the state of an SRAM transistor. Due to the nature of an SRAM device, any fault occurring in one of its transistors gets latched, resulting in a higher probability of an error (when compared with glitches in logic transistors getting masked if the glitch does not occur close to the clock edge). As such, we assume $P'_e = 100P_e$. Putting it all together, we have $P_{R_i} = 1 - P_1 * P_2$.

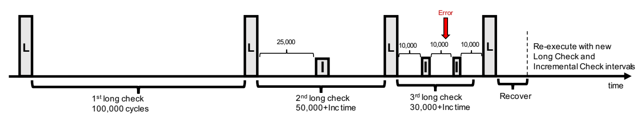
$P_{LOAD\ X}$ Probability of at least 1 error being present in the loaded data of address X. This is analogous to P_{R_i}, with the extended $StateTable$ storing an entry for each cache line. As we assume a perfect off-chip (ECC protected) main memory, cache miss repairs are initialized with a zero probability in error, and cache replacement victims' entries are also evicted from the $StateTable$. Finally, $P_{LOAD\ X}$ encapsulates the probability of an error in the implicit computation of the address X itself (from its base and offset) during the execution of the load, in addition to the probability of an error in the loaded data from the cache line.

P_{SC} Probability of at least 1 error occurring in a sub-core from the last time it was checked. To illustrate, consider the following add instruction: $ADD\ R_3, R_2, R_1$. Then, at the end of instruction, $P_{SC} = 1 - (1 - P_{add})(1 - P_{R_2})(1 - P_{R1})$.

P^1_C Probability of exactly 1 error occurring in a RRNS core from the last time it was checked. This trans-



(a) No error is detected



(b) An error is detected

Figure 5: Error Interval Heuristics(EIH) Mechanism Examples

lates to exactly 1 sub-core being in error (where the sub-core error itself may be of multi-bit form; RRNS can tolerate multi-bit flips within a single residue). Therefore, $P_C = 6C_1 \times P_{SC}(1 - P_{SC})^5$, where the combinatorial choose operator nC_r enumerates the number of ways in which r items can be chosen from n distinct items.

P_C^0 Probability of no error in a RRNS core from the last time it was checked. $P_C^0 = 6C_0 \times (1 - P_{SC})^6 = (1 - P_{SC})^6$.

P_C^{fail} Probability of a RRNS core failing at any given cycle, since the last time it was checked. The current version of the RRNS checkpoint micro-architecture could detects one or two errors. For the (4,1)-RRNS version, we deem ≥ 2 errors in the core as amounting to a failure. Therefore, $P_C^{fail} = \sum_{2 \leq r \leq 6} 6C_r \times P_{SC}^r (1 - P_{SC})^{6-r} = 1 - P_C^0 - P_C^1$. Similarly, for the (4,2)-RRNS version, we deem ≥ 3 errors in the core as amounting to a failure. Therefore, $P_C^{fail} = 1 - P_C^0 - P_C^1 - P_C^2$.

Note that the computation of these error probabilities is done after every instruction (irrespective of the check insertion strategy) for the purposes of bookkeeping such as $StateTable$ update and to estimate the probability of a failure $P_{C,i}^{fail}$ at each time step t_i. We use a typically used reliability metric, Mean Time Between Failure (MTBF) [23], which can be defined as follows: $MTBF = \frac{Total\ Cycles}{CPU\ Frequency\ \times\ \sum_i P_{C,i}^{fail}}$. The subscript i in $P_{C,i}^{fail}$ represents the i^{th} instruction of the instruction stream. MTBF also corresponds to mean time to checkpoint recovery.

5. EXPERIMENTAL EVALUATION

5.1 Minimal Signal Energy with Reasonable Reliability

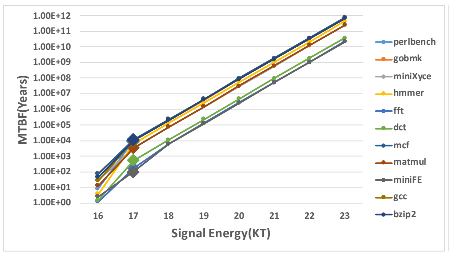
The lowest signal energy values of normal logic in (4,2)-RRNS Single Error Correction(SEC) system are range from 28-31KT [11]. In theory, the related signal energy values could be further turn down by adopting the (4,2)-RRNS Double Error Detectction(DED) configuration. The DED system equips with stronger error detection and recovery ability and only the cases of 3 or more errors are defined as failure. So this give the potential of signal energy goes down further.

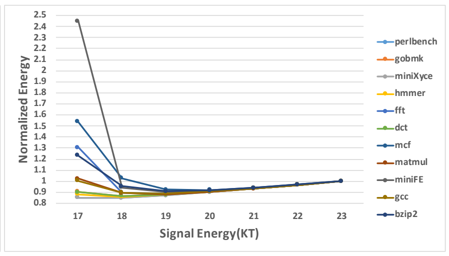
In order to figure out the optimal signal energy of normal logic in energy and Energy Delay Product(EDP), we design 3 signal energy experiments with fixed 100K-cycle long time interval and 10K-cycle incremental interval. Figure 6a gives the MTBF values with signal energy between 16KT to 23KT. The MTBF is defined as 'Reasonable' only if the value is equal or larger than 100 years. From this figure we can reach a conclusion that the minimal signal energy is 17KT to meet the reliability requirement. Figure 6b shows the normalized energy values with signal energy between 17KT-23KT. The energy consumption of 23KT configurations are normalized to 1. If the signal energy is equal or less than 17KT, the energy overhead increase dramatically due to error rate raise. The minimal energy values are detected when the signal energy is set to 18 or 19 KT, depending on the specific workload. Similarly, Figure 6c demonstrates we can get the optimal EDP if the signal energy is set to 18 or 19 KT. Based on these 3 figures, the signal energy could be further turned down to 18-19 KT for (4,2)-RRNS DED configuration, which is much lower than 28-31KT in (4,2)-RRNS Single Error Correction(SEC) system. So the checkpoint and rollback method has the potential to beat the correction scheme. 18 or 19 KT is used in the following experiments for (4,2)-RRNS general logic when making the comparison with other cores.

5.2 Energy Delay Product(EDP) Comparison of 3 RRNS Configurations

The potential of energy and EDP reduction has been verified in Section 5.1. However, figuring out how much optimization it can be achieved is also necessary. Figure 7 shows the Energy Delay Product(EDP) comparison of (4,2)-RRNS Single Error Correction(SEC, CREEPY), (4,1)-RRNS Single Error Detection(SED) and (4,2)-RRNS Double Error Detectcion(DED). And the EDPs of CREEPY are normalized to 1. For the (4,2)-RRNS DED, we used static checkpoint scheme with 40 different configurations for each of workloads. The '*static*' means both *long_check_interval* and *incremental_interval* are stable during the execution. In these 40 configurations, the long check intervals are range from 1K to 500K cycles while the incremental intervals are set from 1K to 100K cycles. The *incremental_intervals* are always less than or equal to *long_check_interval*. The (4,2)-RRNS-static-ave in the figure represents the arithmetic average of the 40 results.

(4,1)-RRNS SED is worst in all the workloads even though it uses less subcore hardware and RIU consistent check logics. The primary reason of this counterintu-


(a) Minimal MTBF Values


(b) Minimal Energy Values

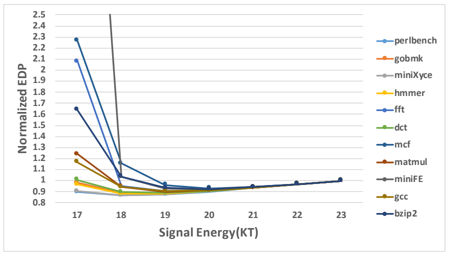

(c) Minimal EDP Values

Figure 6: Minimal Signal Energy Of Normal Logics For All The Workloads

itive phenomenon is (4,1)-RRNS SED loss reliability while trying to reduce the energy consumption in normal computation and checking. On average, one third of errors are undetectable in (4,1)-RRNS SED and this could explain the system MTBFs decreasing. In order to make a fair comparison of all the proposed RRNS configurations, we set the signal energy values and keep the related MTBFs to a same level(Larger or equal to 100 years). In the (4,1)-RRNS SED, we have to use higher single energy to keep the reliability(41-42KT in (4,1)-RRNS SED; 28-31KT in ((4,2)-RRNS SEC); 18-19KT in (4,2)-RRNS DED). Because of the good error detection ability in (4,2)-RRNS DED, it gets the best EDP in this comparison. Failure only occurs when two or more residues are in error. 3 or more residues in error may also detectable in some cases(discussed in previous section?). Because only the static checkpoint scheme is used in this section, the potential of more energy and EDP reduction may be achieved if designing some reasonable adaptive checkpoint and rollback methods.

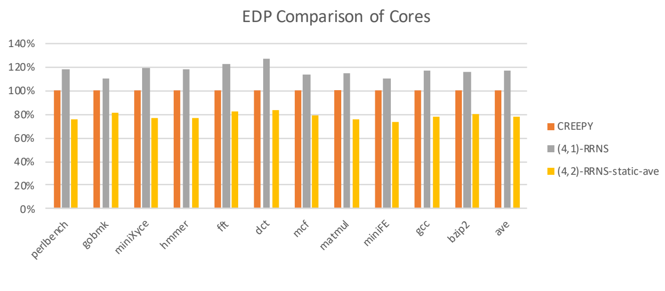


Figure 7: EDP Comparison of Cores

5.3 Energy Breakdown of Static-best Checkpoint Configurations

Figure 8 lists the extra overhead breakdowns of the best static checkpoint configurations for each of the workloads. The '*best*' represents the optimal EDP in 40 configurations. These extra energy overhead are defined into four types: 1) The energy overhead of creating the incremental checkpoints; 2) The energy overhead of all full checkpoint verifications, which operate at the end of every long check interval; 3) Re-execution overhead after system state recovery; 4) Extra recovery energy once the error is detected, and this includes the full checkpoint status and incremental checkpoint status recovery. From the result, the extra energy overhead is low in majority of the workloads. Except gobmk, all the other energy overhead values are less than 6%, which means the static-best configurations do well in energy saving and the potential of adaptive method improvement is limited when comparing with it. But a big problem of static-best is we are difficult to figure out which static checkpoint configuration(value pair of *long_check_interval* and *incremental_interval*) is best for a specific benchmark. We found the best static configuration always vary in different workloads. So efficient adaptive checkpoint methods are still necessary even though they may be no significant improvement when comparing with the best-static checkpoint configuration.

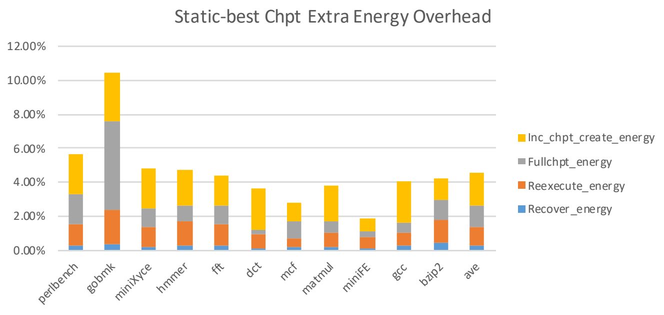


Figure 8: Extra Energy Overhead Breakdown of Static-best Checkpoint Configurations

5.4 Energy and EDP Comparison

Energy is the primary consideration for extending the Denard scaling [24]. Figure 9a compares the energy consumption of all the proposed checkpoint schemes. And the energy consumption of CREEPY, which implements as (4,2)-RRNS Single Error Correction(SEC), is normalized as 1. Because (4,1)-RRNS SED was demonstrated worse than CREEPY in Section 5.2, in this figure we only focus on (4,2)-RRNS Double Error Detection(DED). Static-ave is the average energy values of 40 different static checkpoint point evaluations with the long check intervals vary from 1K to 500K cycles while the incremental intervals are set from 1K to 100K cycles. And Static-best the picks the lowest energy consumption configuration. It should be noted that the related best configuration is inconsistent among the different workloads. In reality it should be hard to get the excellent energy reduction if randomly picking a checkpoint configuration. In other words, the Static-best is hard to achieve in practice. For the Extra Overhead Estimation(EOE) adaptive method, it better than Static-ave in all the benchmarks and worse than Static-best in most cases. The Error Interval Heuristics(EIH) method is always best in all workloads. On average, it gets 31.79%, 8.99%, 1.24% energy reduction respectively when comparing with CREEPY, Static-ave and Static-best. For the EDP comparison in Figure 9b, we can get a similar conclusion, EIH adaptive checking method gets the best EDP and EOE is a little worse than Static-best. The energy and EDP reductions of EIH are not significant when comparing with Static-best. The reason of this limit improvement could be explained in Figure 8. The extra overhead of Static-best is less than 6% in most cases, so the improvement should be constrained to this upper bound. Moreover, the Static-best is always hard to figure out unless running the application multiple times.

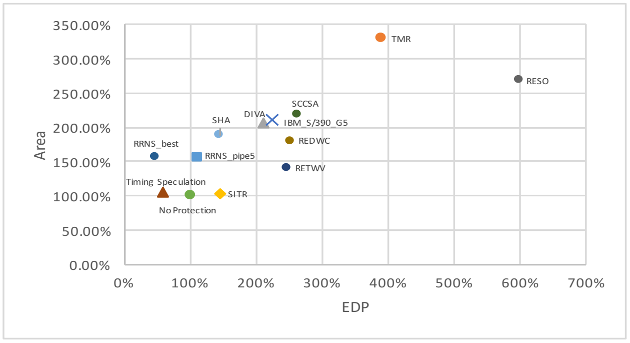
6. RELATED WORK

RNS and RRNS The energy efficient properties of RNS due to its low-bit-width operations and absence of carries across residues has found applications in the digital signal processing (DSP) [25–27] domain. Furthermore, the representability of high bit-width integers as a tuple-of-resides has been leveraged by the cryptography (RSA) [28–30] community. Anderson [13] proposed an architecture and ISA for an RNS co-processor designed to run datapath operations in tandem with a general-purpose processor running binary instructions, where the primary role of the general purpose processor is to handle control flow. The RNS co-processor uses an accumulator based ALU and does not support caching or computational error correction (RRNS). Furthermore, it requires a conversion to binary (and vice-versa) for comparison operations, which is expensive. Clearly, our CREEPY architecture is significantly more efficient. A unique feature of their ISA is their ability to encode instructions targeting two ALUs simultaneously. But this can easily be extended to our architecture and enable such Superscalar-like capabilities if need be.

Chiang et al. [31] provide RNS algorithms for comparison and overflow detection, but assume all bases to be odd and do not consider error correction. Similarly, Preethy et al. [32, 33] integrate index-sum multiplication into RNS, but do not consider its impact on the properties of RRNS bases critical to CREEPY.

Ever since Watson and Hastings [14–16] introduced RRNS as an efficient means for computational error correction, there has been a significant body of research [19, 34–60] that strives to improve upon it. These are orthogonal to CREEPY, and further such algorithmic research can be used to optimize aspects of the core itself, such as the RIU.

Computational Error Correction Standard error correcting codes (ECC) [5] have already been adopted into modern memory systems. These codes accommodate errors occurring in storage and communication/network traffic, but are not able to protect computational logic. The naive approach to computational error correction is triple modular redundancy (TMR) [6], requiring over a 200% overhead in area and energy for single error correcting capability. Several techniques in the form of arithmetic codes such as AN codes [70–75], self-checking [76–82] and self-correcting [83–92] adders and multipliers have since been devised. Orthogonally, proposals employ redundancy at a higher granularity, such as timing speculation (wherein error correction capability is limited to circuit timing violations) [68, 69], partial pipeline replication [66] or checkpoint-rollback-recovery such as those in IBM Power8 processors [93]. While these are more efficient than naive TMR, they come with limitations on their error model, or, their area overheads are still over 100% and/or incur a significant performance penalty, owing to the fact that they leverage temporal redundancy in an effort to minimize area overhead [94]

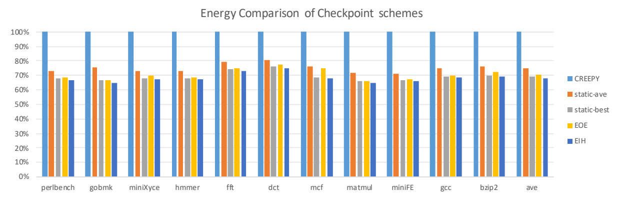


RESO [61], REDWC [62], RETWV [63], SCCSA [64], SHA [65], DIVA [66], SITR [67], Timing Speculation [68, 69]

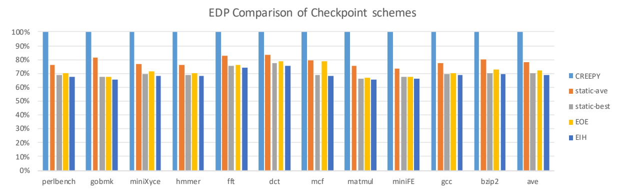
Figure 10: First order comparison of area overhead and energy-delay product (EDP) of various mechanisms for computational error correction, depicting the superiority of RRNS. Computational error correction techniques use a combination of spatial and temporal redundancy techniques. While temporal redundancy allows for a low area overhead, they suffer from a significant performance penalty. Timing speculation techniques seem more efficient than RRNS, however, their error model assumes all bit errors manifest as circuit timing errors, which is not sufficient to work with ultra low energy logic devices.

Figure 10 summarizes some of these techniques in comparison with RRNS. We refer the interested reader to Srikanth et al. [94] for a more detailed survey on some of these non-residue techniques, but the takeaway is that RRNS is generally considered superior in terms of capability and efficiency for computational error resilience.

Approaches that employ timing speculation [68, 69] may seem superior to RRNS at first glance. However, the error model that can be supported by an RRNS error correcting microarchitecture is orthogonal to theirs, if not broader. For example, razor [68] uses conventional transistors, therefore lowering V_{dd} lowers MOS-



(a) Energy



(b) Energy Delay Product(EDP)

Figure 9: Comparison of Checkpoint Schemes

FET switching speed, resulting in a frequency drop, which could cause setup time violations that they handle via a delayed latch mechanism. They assume that any error manifests itself as a timing error. Similarly, decor [69] uses a delayed commit approach (with rollback support) to handle violations in timing margins. However, with emerging devices (Section **??**), V_{dd} can be lowered to few tens of millivolts without frequency loss, meaning that operating at the resultant thermal noise floor leads to *stochastic, intermittent* bit flips, which cannot be captured as circuit timing errors. Unlike such approaches, a CREEPY core can not only tolerate such errors in the data path, but also in the control path between memory accesses.

In terms of being able to tolerate control path errors, approaches such as DIVA [66] that replicate parts of the pipeline are capable. Their design provides recovery by having a simple core recalculate results of an out-of-order core. In this approach, the simple core is assumed to be error-free. This is similar to a "double-modular-redundancy" approach with a rad-hard node, implying a relatively high overhead. Furthermore, if the rad-hard simple core is instead prone to error, checkpoint and re-execute methods would need to be employed, similar to the IBM POWER7/8 processors [93]. On the other hand, a CREEPY core is able to tolerate errors in its redundant as well as non-redundant computations.

7. CONCLUSION

The upcoming next generation device such as tunneling FETs and ferroelectric/negative- capacitance FETs enables reduction of supply voltage to few tens of millivolts without degradation in switching speed. However, as a result of operating close the the kT noise floor, computational logic is subject to intermittent, stochastic errors. The RRNS representation is a promising approach towards using such ultra low power devices, by employing efficient computational error correction or detection with checkpoint. In this paper, we design (4,1)-RRNS and (4,2)-RRNS checkpoint systems with reasonable base sets and supported incremental checkpoint mechanism. By using the RRNS checkpoint and rollback mechanism, we can safely remove a ROM unit which stored the large size error correction table. We also verified that (4,2)-RRNS DED configuration could further reduce energy and EDP while (4,1)-RRNS SED doesn't. Finally we proposed 2 adaptive checkpoint interval adjustment methods and reducing 30-40% of energy and EDP compared with (4,2)-RRNS SEC system(CREEPY).

Acknowledgements

We thank the anonymous reviewers of the review process for their constructive feedback on this research.

8. REFERENCES

[1] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM journal of research and development 5.3*, pp. 183–191, 1961.

[2] M. Neyman, "The negentropy principle in information-processing systems," *Telecommunications and Radio Engineering 2*, 1966.

[3] U. E. Avci, D. H. Morris, and I. A. Young, "Tunnel field-effect transistors: Prospects and challenges," *IEEE Journal of the Electron Devices Society*, vol. 3, no. 3, pp. 88–95, May 2015.

[4] S. Agarwal, J. Cook, E. DeBenedictis, M. P. Frank, G. Cauwenberghs, S. Srikanth, B. Deng, E. R. Hein, P. G. Rabbat, and T. M. Conte, "Energy efficiency limits of logic and memory," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, Oct 2016, pp. 1–8.

[5] F. J. MacWilliams and N. J. A. Sloane, *The theory of error-correcting codes.* Elsevier, 1977.

[6] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata studies*, vol. 34, pp. 43–98, 1956.

[7] D. Lipetz and E. Schwarz, "Self checking in current floating-point units," in *2011 IEEE 20th Symposium on Computer Arithmetic*, July 2011, pp. 73–76.

[8] J. Warnock, Y. Chan, W. Huott, S. Carey, M. Fee, H. Wen, M. J. Saccamango, F. Malgioglio, P. Meaney, D. Plass, Y. H. Chan, M. Mayo, G. Mayer, L. Sigal, D. Rude, R. Averill, M. Wood, T. Strach, H. Smith, B. Curran, E. Schwarz, L. Eisen, D. Malone, S. Weitzel, P. K. Mak, T. McPherson, and C. Webb, "A 5.2ghz microprocessor chip for the ibm zenterprise system," in *2011 IEEE International Solid-State Circuits Conference*, Feb 2011, pp. 70–72.

[9] D. Henderson, B. Warner, and J. Mitchell, "Ibm power6 processor-based systems: Designed for availability," in *White Paper, IBM Corporation*, 2007.

[10] D. Henderson, J. Mitchell, and G. Ahrens, "Power7 system ras: Key aspects of power systems reliability, availability, and serviceability," in *White Paper, IBM Corporation*, 2010.

[11] B. Deng, S. Srikanth, E. R. Hein, T. M. Conte, E. Debenedictis, J. Cook, and M. P. Frank, "Extending moore's law via computationally error-tolerant computing," *ACM Trans. Archit. Code Optim (TACO).*, vol. 15, no. 1, pp. 8:1–8:27, Mar. 2018.

[12] E. B. Olsen, "Introduction of the residue number arithmetic logic unit with brief computational complexity analysis (rez-9 soft processor)," *Whitepaper, Digital System Research*, 2015.

[13] D. Anderson, “Design and implementation of an instruction set architecture and an instruction execution unit for the rez9 coprocessor system,” *M.S. Thesis, U of Nevada LV*, 2014.

[14] C. W. Hastings, “Automatic detection and correction of errors in digital computers using residue arithmetic,” in *Region Six Annu. Conf.* IEEE, 1966, pp. 429–464.

[15] R. W. Watson, “Error detection and correction and other residue interacting operations in a residue redundant number system,” in *Univ. California, Berkeley*, 1965.

[16] R. W. Watson and C. W. Hastings, “Self-checked computation using residue arithmetic,” *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1920–1931, 1966.

[17] B. Deng, S. Srikanth, E. R. Hein, P. G. Rabbat, T. M. Conte, E. DeBenedictis, and J. Cook, “Computationally-redundant energy-efficient processing for y’all (creepy),” in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, Oct 2016, pp. 1–8.

[18] S. Srikanth, P. G. Rabbat, E. R. Hein, B. Deng, T. M. Conte, E. DeBenedictis, J. Cook, and M. P. Frank, “Memory system design for ultra low power, computationally error resilient processor microarchitectures,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 696–709.

[19] O. Goldreich, D. Ron, and M. Sudan, “Chinese remaindering with errors,” in *Proceedings of the thirty-first annual ACM symposium on Theory of computing.* ACM, 1999, pp. 225–234.

[20] N. Naksinehaboon, Y. Liu, C. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, “Reliability-aware approach: An incremental checkpoint/restart model in hpc environments,” in *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, May 2008, pp. 783–788.

[21] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, “Adaptive incremental checkpointing for massively parallel systems,” in *Proceedings of the 18th Annual International Conference on Supercomputing*, ser. ICS ’04. New York, NY, USA: ACM, 2004, pp. 277–286. [Online]. Available: http://doi.acm.org/10.1145/1006209.1006248

[22] H. Li, L. Pang, and Z. Wang, “Two-level incremental checkpoint recovery scheme for reducing system total overheads,” in *PLoS ONE*, vol. 9, no. 8, 2014, p. e104591.

[23] J. Tan and O. Rosen, “Process for determining competing cause event probability and/or system availability during the simultaneous occurrence of multiple events,” Apr. 22 2004, uS Patent App. 10/272,156. [Online]. Available: https://www.google.com/patents/US20040078167

[24] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted mosfet’s with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[25] R. Chokshi, K. S. Berezowski, A. Shrivastava, and S. J. Piestrak, “Exploiting residue number system for power-efficient digital signal processing in embedded processors,” in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems.* ACM, 2009, pp. 19–28.

[26] E. D. Di Claudio, F. Piazza, and G. Orlandi, “Fast combinatorial rns processors for dsp applications,” *IEEE transactions on computers*, vol. 44, no. 5, pp. 624–633, 1995.

[27] J. Ramirez, A. Garcia, S. Lopez-Buedo, and A. Lloris, “Rns-enabled digital signal processor design,” *Electronics Letters*, vol. 38, no. 6, pp. 266–268, 2002.

[28] J.-C. Bajard and L. Imbert, “A full rns implementation of rsa,” *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 769–774, 2004.

[29] C. Y. Hung and B. Parhami, “Fast rns division algorithms for fixed divisors with application to rsa encryption,” *Information Processing Letters*, vol. 51, no. 4, pp. 163–169, 1994.

[30] S.-M. Yen, S. Kim, S. Lim, and S.-J. Moon, “Rsa speedup with chinese remainder theorem immune against hardware fault cryptanalysis,” *IEEE Transactions on computers*, vol. 52, no. 4, pp. 461–472, 2003.

[31] J.-S. Chiang and M. Lu, “Floating-point numbers in residue number systems,” *Computers & Mathematics with Applications*, vol. 22, no. 10, pp. 127–140, 1991.

[32] A. Preethy and D. Radhakrishnan, “A 36-bit balanced moduli mac architecture,” in *Circuits and Systems, 1999. 42nd Midwest Symposium on*, vol. 1. IEEE, 1999, pp. 380–383.

[33] ——, “Rns-based logarithmic adder,” *IEE Proceedings-Computers and Digital Techniques*, vol. 147, no. 4, pp. 283–287, 2000.

[34] T. F. Tay and C.-H. Chang, “A non-iterative multiple residue digit error detection and correction algorithm in rrns,” *IEEE transactions on computers*, vol. 65, no. 2, pp. 396–408, 2016.

[35] J.-C. Bajard, J. Eynard, and N. Merkiche, “Multi-fault attack detection for rns cryptographic architecture,” in *Computer Arithmetic (ARITH), 2016 IEEE 23nd Symposium on.* IEEE, 2016, pp. 16–23.

[36] H. Xiao, H. K. Garg, J. Hu, and G. Xiao, “New error control algorithms for residue number system codes,” *ETRI Journal*, vol. 38, no. 2, pp. 326–336, 2016.

[37] L. Xiao and X.-G. Xia, “Error correction in polynomial remainder codes with non-pairwise coprime moduli and robust chinese remainder theorem for polynomials,” *IEEE Transactions on Communications*, vol. 63, no. 3, pp. 605–616, 2015.

[38] C.-H. Chang, A. S. Molahosseini, A. A. E. Zarandi, and T. F. Tay, “Residue number systems: A new paradigm to datapath optimization for low-power and high-performance digital signal processing applications,” *IEEE circuits and systems magazine*, vol. 15, no. 4, pp. 26–44, 2015.

[39] T. F. Tay and C.-H. Chang, “A new algorithm for single residue digit error correction in redundant residue number system,” in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on.* IEEE, 2014, pp. 1748–1751.

[40] P. Yin and L. Li, “A new algorithm for single error correction in rrns,” in *Communications, Circuits and Systems (ICCCAS), 2013 International Conference on*, vol. 2. IEEE, 2013, pp. 178–181.

[41] H.-Y. Lo and T.-W. Lin, “Parallel algorithms for residue scaling and error correction in residue arithmetic,” *Wireless Engineering and Technology*, vol. 4, no. 04, p. 198, 2013.

[42] A. Sengupta and B. Natarajan, “Performance of systematic rrns based space-time block codes with probability-aware adaptive demapping,” *IEEE Transactions on Wireless Communications*, vol. 12, no. 5, pp. 2458–2469, 2013.

[43] N. Z. Haron and S. Hamdioui, “Redundant residue number system code for fault-tolerant hybrid memories,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 7, no. 1, p. 4, 2011.

[44] Y. Tang, E. Boutillon, C. Jégo, and M. Jézéquel, “A new single-error correction scheme based on self-diagnosis residue number arithmetic,” in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on.* IEEE, 2010, pp. 27–33.

[45] V. T. Goh and M. U. Siddiqi, “Multiple error detection and correction based on redundant residue number systems,” *IEEE Transactions on Communications*, vol. 56, no. 3, 2008.

[46] A. Sweidan and A. A. Hiasat, “On the theory of error control based on moduli with common factors,” *Reliable computing*, vol. 7, no. 3, pp. 209–218, 2001.

[47] R. S. Katti, “A new residue arithmetic error correction scheme,” *IEEE transactions on computers*, vol. 45, no. 1, pp. 13–19, 1996.

[48] H. Krishna, B. Krishna, K.-Y. Lin, and J.-D. Sun, *Computational Number Theory and Digital Signal Processing: Fast Algorithms and Error Control Techniques.* CRC Press, 1994, vol. 6.

[49] E. D. Di Claudio, G. Orlandi, and F. Piazza, “A systolic

redundant residue arithmetic error correction circuit," *IEEE Transactions on Computers*, vol. 42, no. 4, pp. 427–432, 1993.

[50] H. Krishna, K.-Y. Lin, and J.-D. Sun, "A coding theory approach to error control in redundant residue number systems. i. theory and single error correction," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 1, pp. 8–17, 1992.

[51] J.-D. Sun and H. Krishna, "A coding theory approach to error control in redundant residue number systems. ii. multiple error detection and correction," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 1, pp. 18–34, 1992.

[52] J.-D. Sun, H. Krishna, and K. Lin, "A superfast algorithm for single-error correction in rrns and hardware implementation," in *Circuits and Systems, 1992. ISCAS'92. Proceedings., 1992 IEEE International Symposium on*, vol. 2. IEEE, 1992, pp. 795–798.

[53] G. A. Orton, L. E. Peppard, and S. E. Tavares, "New fault tolerant techniques for residue number systems," *IEEE transactions on computers*, vol. 41, no. 11, pp. 1453–1464, 1992.

[54] C.-C. Su and H.-Y. Lo, "An algorithm for scaling and single residue error correction in residue number systems," *IEEE Transactions on Computers*, vol. 39, no. 8, pp. 1053–1064, 1990.

[55] V. Ramachandran, "Single residue error correction in residue number systems," *IEEE transactions on computers*, vol. 32, no. 5, pp. 504–507, 1983.

[56] M. Etzel and W. Jenkins, "Redundant residue number systems for error detection and correction in digital filters," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 5, pp. 538–545, 1980.

[57] F. Barsi and P. Maestrini, "Error detection and correction by product codes in residue number systems," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 915–924, 1974.

[58] S.-S. Yau and Y.-C. Liu, "Error correction in redundant residue number systems," *IEEE Transactions on Computers*, vol. 100, no. 1, pp. 5–11, 1973.

[59] T. R. Rao, "Biresidue error-correcting codes for computer arithmetic," *IEEE Transactions on computers*, vol. 100, no. 5, pp. 398–402, 1970.

[60] N. S. Szabo and R. I. Tanaka, *Residue arithmetic and its applications to computer technology.* McGraw-Hill, 1967.

[61] J. H. Patel and L. Y. Fung, "Concurrent error detection in alu's by recomputing with shifted operands," *IEEE Trans. Computers*, vol. 31, no. 7, pp. 589–595, 1982.

[62] B. W. Johnson, J. H. Aylor, and H. H. Hana, "Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit vlsi adder," *journal of solid-state circuits*, vol. 23, no. 1, pp. 208–215, 1988.

[63] Y.-M. Hsu and E. Swartzlander, "Time redundant error correcting adders and multipliers," in *Defect and Fault Tolerance in VLSI Systems, Proceedings., International Workshop on.* IEEE, 1992, pp. 247–256.

[64] D. P. Vasudevan, P. K. Lala, and J. P. Parkerson, "Self-checking carry-select adder design based on two-rail encoding," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 12, pp. 2696–2705, 2007.

[65] S. Peng and R. Manohar, "Fault tolerant asynchronous adder through dynamic self-reconfiguration," in *Computer Design: VLSI in Computers and Processors, ICCD. Proceedings.International Conference on.* IEEE, 2005, pp. 171–178.

[66] T. M. Austin, "Diva: A reliable substrate for deep submicron microarchitecture design," in *Microarchitecture, MICRO-32. Proceedings. 32nd Annual International Symposium on.* IEEE, 1999, pp. 196–207.

[67] E. Mizan, T. Amimeur, and M. F. Jacome, "Self-imposed temporal redundancy: An efficient technique to enhance the reliability of pipelined functional units," in *Computer Architecture and High Performance Computing, SBAC-PAD. 19th International Symposium on.* IEEE, 2007, pp. 45–53.

[68] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner *et al.*, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Microarchitecture,MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on.* IEEE, 2003, pp. 7–18.

[69] M. S. Gupta, K. K. Rangan, M. D. Smith, G.-Y. Wei, and D. Brooks, "Decor: A delayed commit and rollback mechanism for handling inductive noise in processors," in *High Performance Computer Architecture, HPCA, IEEE 14th International Symposium on.* IEEE, 2008, pp. 381–392.

[70] D. T. Brown, "Error detecting and correcting binary codes for arithmetic operations," *IRE Transactions on Electronic Computers*, no. 3, pp. 333–337, 1960.

[71] C. Fetzer, U. Schiffel, and M. Süßkraut, "An-encoding compiler: Building safety-critical systems with commodity hardware," in *International Conference on Computer Safety, Reliability, and Security.* Springer, 2009, pp. 283–296.

[72] P. Forin, "Vital coded microprocessor principles and application for various transit systems," *IFAC Control, Computers, Communications*, pp. 79–84, 1989.

[73] C.-K. Liu, "Error-correcting-codes in computer arithmetic," DTIC Document, Tech. Rep., 1972.

[74] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer, "Anb-and anbdmem-encoding: detecting hardware errors in software," in *International Conference on Computer Safety, Reliability, and Security.* Springer, 2010, pp. 169–182.

[75] U. Wappler and C. Fetzer, "Hardware failure virtualization via software encoded processing," in *Industrial Informatics, 2007 5th IEEE International Conference on*, vol. 2. IEEE, 2007, pp. 977–982.

[76] B. W. Johnson, J. H. Aylor, and H. H. Hana, "Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit vlsi adder," *IEEE journal of solid-state circuits*, vol. 23, no. 1, pp. 208–215, 1988.

[77] O. Keren, I. Levin, V. Ostrovsky, and B. Abramov, "Arbitrary error detection in combinational circuits by using partitioning," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on.* IEEE, 2008, pp. 361–369.

[78] D. Marienfeld, E. S. Sogomonyan, V. Ocheretnij, and M. Gossel, "New self-checking output-duplicated booth multiplier with high fault coverage for soft errors," in *Test Symposium, 2005. Proceedings. 14th Asian.* IEEE, 2005, pp. 76–81.

[79] M. Nicolaidis, "Carry checking/parity prediction adders and alus," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 1, pp. 121–128, 2003.

[80] M. Nicolaidis and R. Duarte, "Design of fault-secure parity-prediction booth multipliers," in *Design, Automation and Test in Europe, 1998., Proceedings.* IEEE, 1998, pp. 7–14.

[81] M. Nicolaidis and H. Bederr, "Efficient implementations of self-checking multiply and divide arrays," in *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings.* IEEE, 1994, pp. 574–579.

[82] D. P. Vasudevan and P. K. Lala, "A technique for modular design of self-checking carry-select adder," in *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on.* IEEE, 2005, pp. 325–333.

[83] Y. Sun, M. Zhang, S. Li, and Y. Zhao, "Cost effective soft error mitigation for parallel adders by exploiting inherent redundancy," in *IC Design and Technology (ICICDT), 2010 IEEE International Conference on.* IEEE, 2010, pp. 224–227.

[84] Y.-M. Hsu and E. Swartzlander, “Time redundant error correcting adders and multipliers,” in *Defect and Fault Tolerance in VLSI Systems, 1992. Proceedings., 1992 IEEE International Workshop on.* IEEE, 1992, pp. 247–256.

[85] S. Peng and R. Manohar, “Fault tolerant asynchronous adder through dynamic self-reconfiguration,” in *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on.* IEEE, 2005, pp. 171–178.

[86] S. Dolev, S. Frenkel, D. E. Tamir, and V. Sinelnikov, “Preserving hamming distance in arithmetic and logical operations,” *Journal of Electronic Testing*, vol. 29, no. 6, pp. 903–907, 2013.

[87] S. Ghosh, P. Ndai, and K. Roy, “A novel low overhead fault tolerant kogge-stone adder using adaptive clocking,” in *Design, Automation and Test in Europe, 2008. DATE’08.* IEEE, 2008, pp. 366–371.

[88] E. Krekhov, A.-r. A. Pavlov, A. Pavlov, P. Pavlov, D. Smirnov, A. Tsar’kov, P. Chistopol’skii, A. Shandrikov, B. Sharikov, and D. Yakimov, “A method of monitoring execution of arithmetic operations on computers in computerized monitoring and measuring systems,” *Measurement Techniques*, vol. 51, no. 3, pp. 237–241, 2008.

[89] J. Mathew, S. Banerjee, P. Mahesh, D. Pradhan, A. Jabir, and S. Mohanty, “Multiple bit error detection and correction in gf arithmetic circuits,” in *Electronic System Design (ISED), 2010 International Symposium on.* IEEE, 2010, pp. 101–106.

[90] W. Rao and A. Orailoglu, “Towards fault tolerant parallel prefix adders in nanoelectronic systems,” in *Design, Automation and Test in Europe, 2008. DATE’08.* IEEE, 2008, pp. 360–365.

[91] W. Rao, A. Orailoglu, and R. Karri, “Fault identification in reconfigurable carry lookahead adders targeting nanoelectronic fabrics,” in *Test Symposium, 2006. ETS’06. Eleventh IEEE European.* IEEE, 2006, pp. 63–68.

[92] M. Valinataj and S. Safari, “Fault tolerant arithmetic operations with multiple error detection and correction,” in *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT’07. 22nd IEEE International Symposium on.* IEEE, 2007, pp. 188–196.

[93] IBM, “Ibm power system e880 server, an ibm power8 technology-based system, addresses the requirements of an industry-leading enterprise class system,” 2014.

[94] S. Srikanth, B. Deng, and T. M. Conte, “A brief survey of non-residue based computational error correction,” *arXiv preprint arXiv:1611.03099*, 2016.

APPENDIX

.1 Expected Error Time in EOE Adaptive Checking

The expected error time in the last long check interval could be defined as follow:

$$E(X) = \lim_{N\to\infty} \frac{\mathit{Sum\ of\ Error\ Cycles\ in\ N\ Experiments}}{\mathit{Num\ of\ Errors\ in\ N\ Experiments}}$$

$$= \lim_{N\to\infty} \frac{N*(1*P_1+2*P_2+3*P_3...+L*P_L)}{N*(P_1+P_2+P_3...+P_L)}$$

$$= \frac{(1*P_1+2*P_2+3*P_3...+L*P_L)}{(P_1+P_2+P_3...+P_L)} \qquad (1)$$

P_i in Formula (1) represents the the error is detected in the i_{th} cycle of the last long interval and $P_i = A(1-A)^{i-1}$. We assume the system issues a load instruction in every cycle and make the expected value estimation in a simple way. So we have

$$A = 1-(1-P_{err\_sram\_t})^{TCount}$$

$P_{err\_sram\_t}$ represents the fault probability of a single SRAM transistor and $TCount$ is the active SRAM transistor number of the load operation. So we get a new format of formula (1):

$$E(X) = \frac{\sum_{i=1}^{L} i*A(1-A)^{i-1}}{\sum_{i=1}^{L} A(1-A)^{i-1}} \qquad (2)$$

For the denominator of Formula (2):

$$\sum_{i=1}^{L} A(1-A)^{i-1} = \sum_{i=1}^{L}(1-A)^{i-1} - \sum_{i=1}^{L}(1-A)^{i}$$

$$= \frac{1-(1-A)^L}{1-(1-A)} - \frac{(1-A)(1-(1-A)^L)}{1-(1-A)}$$

$$= \frac{(1-A)^{L+1}-(1-A)^L+A}{A} \qquad (3)$$

Similarly, for the numerator of Formula (2):

$$\sum_{i=1}^{L} i*A(1-A)^{i-1} = \sum_{i=1}^{L} i*(1-A)^{i-1} - \sum_{i=1}^{L} i*(1-A)^{i} \qquad (4)$$

Define a variable T:

$$T = \sum_{i=1}^{L} i*(1-A)^i = (1-A)+2(1-A)^2+\cdots+L(1-A)^L \qquad (5)$$

And then multiply *(1-A)* in both sides of Formula (5):

$$(1-A)T = (1-A)^2+2(1-A)^3+\cdots+L(1-A)^{L+1} \qquad (6)$$

Formula (5) subtracts Formula (6):

$$T-(1-A)T = (1-A)+(1-A)^2+\cdots+(1-A)^L - L(1-A)^{L+1}$$

$$= \frac{1-A-(1-A)^{L+1}}{A} - L(1-A)^{L+1} \qquad (7)$$

From Formula (7), we can compute the value of T:

$$T = \frac{1}{1-(1-A)} * (\frac{1-A-(1-A)^(L+1)}{A} - L(1-A)^{L+1})$$

$$= \frac{1-A-(1-A)^{L+1}}{A^2} - \frac{L(1-A)^{L+1}}{A} \qquad (8)$$

$$\therefore \mathit{Formula\ (4)} = (\frac{1}{1-A}-1)*T = \frac{1-(1-A)^L}{A} - L(1-A)^L \qquad (9)$$

Finally, we put Formula (3) and (9) into Formula (2), and get the value of E(X):

$$E(X) = (2) = [\frac{1-(1-A)^L}{A} - L(1-A)^L] * \frac{A}{(1-A)^{L+1}-(1-A)^L+A}$$

$$= \frac{1-(1-A)^L - AL(1-A)^L}{(1-A)^{L+1}-(1-A)^L+A} = \frac{1-(1+AL)(1-A)^L}{A-A(1-A)^L} \qquad (10)$$