Computationally-Redundant Energy-Efficient Processing for Y'all (CREEPY)

Bobin Deng*, Sriseshan Srikanth*, Eric R. Hein†,
Paul G. Rabbat† and Thomas M. Conte*†
*School of Computer Science
†School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA
Email: {bdeng, seshan, heine, prabbat3, conte}@gatech.edu

Erik DeBenedictis‡ and Jeanine Cook‡
‡Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, USA
Email: {epdeben, jeacook}@sandia.gov

***Abstract*—Dennard scaling has ended. Lowering the voltage supply (Vdd) to sub volt levels causes intermittent losses in signal integrity, rendering further scaling (down) no longer acceptable as a means to lower the power required by a processor core. However, if it were possible to recover the occasional losses due to lower Vdd in an efficient manner, one could effectively lower power. In other words, by deploying the right amount and kind of redundancy, we can strike a balance between overhead incurred in achieving reliability and savings realized by permitting lower Vdd. One promising approach is the Redundant Residue Number System (RRNS) representation. Unlike other error correcting codes, RRNS has the important property of being closed under addition, subtraction and multiplication. Thus enabling correction of errors caused due to both faulty storage and compute units. Furthermore, the incorporated approach uses a fraction of the overhead and is more efficient when compared to the conventional technique used for compute-reliability.**

In this article, we provide an overview of the architecture of a CREEPY core that leverages this property of RRNS and discuss associated algorithms such as error detection/correction, arithmetic overflow detection and signed number representation. Finally, we demonstrate the usability of such a computer by quantifying a performance-reliability trade-off and provide a lower bound measure of tolerable input signal energy at a gate, while still maintaining reliability.

I. Introduction

Dennard scaling [6] has been one of the main phenomena driving efficiency improvements of computers through several decades. The main idea of this law is that transistors consume the same amount of power per unit area as they scale down in size. However, leakage current and threshold voltage limits have caused Dennard scaling to end [2]. This essentially negates any performance benefits that Moore's law may provide in the future; power considerations dictate that a higher transistor density results in either a lower clock rate or a reduction in active chip area.

In this paper, we propose a scalable architectural technique to effectively extend the performance benefits of Moore's law. We enable reducing the supply voltage beyond conservative thresholds by efficiently correcting intermittent computational errors that may arise from doing so. Energy benefits are observed as long as the overhead incurred in error correction is less than that saved by lowering V_{dd}. Furthermore, it may also be possible to lower energy by using *marginal*/post-CMOS devices, but such devices may sometimes be unreliable (due to tunneling effects, for instance [1]). The CREEPY approach of introducing error correcting hardware to lower energy is therefore deemed beneficial.

Figure 1 depicts a prior study conducted by one of the authors that estimates the potential of such a CREEPY computing paradigm in the future.

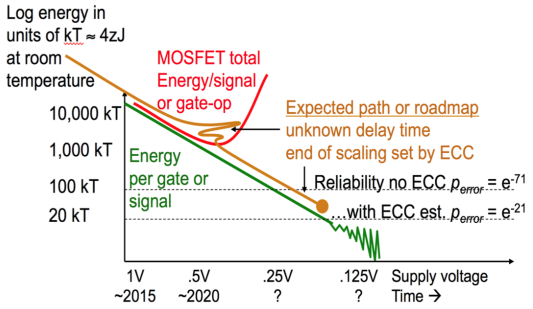


Fig. 1: Energy consequences of the fact that the error probability increases exponentially with decrease in signal energy. Here, the term ECC encapsulates any error correction mechanism in general.

We first provide some mathematical background that forms the basis of our proposed architecture.

II. Background

A. Triple Modular Redundancy (TMR)

The conventional approach to computational fault tolerance is TMR [9]; the idea is to replicate the hardware twice (for a sum total of three computations per computation) and then take a majority vote. With a model that assumes that at most one of these three computations can be in error at any given point in time, it follows that at least two of the computations are error-free; this can thus be used to detect and correct a single error, assuming an error-free voter.

While simple to understand and implement, this introduces an overhead of 200% in area and power, which leaves plenty of room for improvement. Any energy savings from lowering

978-1-5090-1370-8/16/$31.00 © 2016 IEEE

TABLE I: 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 (residues) function independently of one another.					
	An error in any one of these columns (residues) can be corrected by the remaining columns.					

V_{dd} and/or using post-CMOS devices would be eclipsed due to this overhead in correcting resultant errors.

B. Residue Number System (RNS)

The Residue Number System [7] has been used as an alternative to the binary number system chiefly to speed up computation, especially in signal processors [4], [5], [11]. 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 (with the exception of division, comparison and binary bit manipulation). We present some of the properties of an RNS system without proof.

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 an RNS system that is defined by the set of bases B. Specifically, for x such that $x \in \mathbb{N},\ x < M$, then, $x \equiv (|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, ..., |x|_{m_n})$, 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 $x, y \in \mathbb{N},\ x, y < M$, we have $|x\ op\ y|_m = ||x|_m\ op\ |y|_m|_m$, where op is any add/subtract/multiply operation. Unsupported arithmetic operations (including division), thereby, would incur a performance and energy overhead; such operations should be carefully handled by the compiler, the specifics of which warrant further research and are beyond the scope of this paper.

C. Redundant Residue Number System (RRNS)

To augment RNS with fault tolerance, r redundant bases are introduced [8], [12], [13]. 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 an *error code* because of the fact that all residues are transformed in an identical manner under arithmetic operations. For x such that $x \in \mathbb{N},\ x < M$, then, $x \equiv (|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, ..., |x|_{m_n}, |x|_{m_{n+1}}, ..., |x|_{m_r})$ contains n non-redundant residues as well as r redundant residues.

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 I provides a simple example, Section IV-D outlines necessary algorithms to do the single error correction. Research by Watson and Hastings [8], [12], [13] lays the foundation for the underlying theoretical framework that is used and extended in our work. We use (199, 233, 194, 239, 251, 509) as our (4, 2)-RRNS system, providing a range $M = 199 \times 233 \times 194 \times 239 \in (2^{31}, 2^{32})$. These RRNS moduli were chosen to fit in 8-bit or 9-bit registers.

It can be seen that a redundant residue number system achieves a higher efficiency due to enhanced bit-level parallelism while also providing resilience with only 50% overhead. As the granularity of an error is that of an entire residue, RRNS is capable of potentially correcting multi-bit errors as well. Moreover, RRNS could efficiently handle the error chaining problem; if we are summing up an array and a single add is in error, we can fix the final sum at the end with a single check.

We now design a computer using these properties.

III. CREEPY Overview

Given the potential of *marginal* devices, *i.e.,* post-CMOS/millivolt switches that are conducive to lowering voltages (reliability deprecates, but gracefully), CREEPY aims to achieve lower energy in high throughput exascale HPC systems by lowering the supply voltage while efficiently correcting the resulting errors.

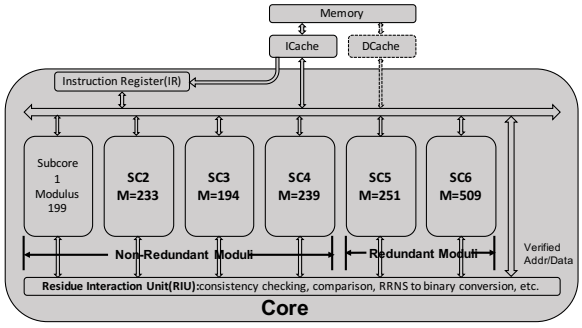


Fig. 2: The CREEPY Core with the reference RRNS system. The register file and data cache are distributed across subcores.

A CREEPY core consists of 6 *subcores*, an Instruction Register (IR) and a Residue Interaction Unit (RIU), as depicted in Figure 2.

Each subcore is fault-isolated from the others because it is designed to operate on a single residue of data. This can be thought of as analogous to a bit-slice processor. After posting a successful instruction fetch (the instruction cache stores instructions in binary, and is ECC protected), the checked instruction is dispatched onto the 6 subcores, which then proceed to operate on their corresponding slice of data. For example, adding two registers is done on a per residue basis; the register file is itself distributed across the 6 subcores. Similarly, the data cache is also distributed across the 6 subcores and stores RRNS protected data. The RIU is then responsible to perform any operations that involve more than a single residue, such as RRNS consistency checking, comparison and conversion to binary.

CREEPY employs standard ECC[10] to protect the main memory because of ECC's compactness and efficiency when it comes to protecting stored data. As such, both data and instructions are simply 32 bits each, not counting their ECC protection. However, the 32 bit representation of data is in RNS form (as opposed to binary). The memory controller checks ECC on a processor load and generates the two redundant residues before loading data into the last level cache. Similarly, it generates ECC upon a processor store (and the redundant residues are not stored into the main memory).

The focus of this paper is on the architecture of a computationally error tolerant core/processing element and *not* on ECC techniques for reliable storage.

IV. CREEPY Core Design

In this section, we present several aspects of a CREEPY core design.

A. Instruction Set Architecture(ISA)

In order to simplify instruction fetch and decode stages, all instructions are fixed to 32 bits wide. The ISA expects 32 registers (R0-R31), with R0 hard-wired to zero, R30 being the link register and R31 storing the default next PC ($R31 = PC + 4$). In our micro-architecture, each register is 49 bits long (*i.e.,* it contains the RRNS redundant residues as well) and is sliced on a per-modulus (sub-core) basis. The data cache is also implemented in a similar manner, as it stores data in an RRNS format. We discuss the formats of several important CREEPY instructions in the remaining part of this section.

1) R-Format (ADD/SUB/MUL)
These instructions assume that the destination operand as well as both source operands are registers.

Opcode	Src Reg1	Src Reg2	Dest Reg	Reserved
6b	5b	5b	5b	11b

2) I-Format (ADDI/SUBI/MULI)
For instructions that require compiler generated immediate literals, two new instructions (that always occur in succession without exception) are defined. Telescopic op-codes are employed to facilitate implementation of such *set* instructions. The fundamental need for the *set* instruction arises from the fact that literals are 49 bit RRNS values and would not otherwise simply fit within a 32 bit field (next to an immediate instruction, for example).
Set123 sets the the first 3 residues of the immediate value into the first 3 sub-core slices of the destination register and *Set456* sets the remaining 3 residues of the immediate value into the other three sub-core slices of the destination register.

Opcode	Dest	Reserved	Residue3	Residue2	Residue1
11[2b]	5b	0[1b]	8b	8b	8b

Opcode	Dest	Residue6	Residue5	Residue4
11[2b]	5b	9b	8b	8b

For an example, consider the immediate instruction *Addi R1, R2, 0x020202020202*. A CREEPY program would implement this instruction as follows:

a) Set123 R3, 020202
b) Set456 R3, 020202
c) Add R1, R3, R2

3) Branch

Opcode	Reg1	Reg2	Reg3	Link	Reserved
6b	5b	5b	5b	1b	10b

Recall that R0 = 0, R31 = PC + 4 and that R30 is the link register. A CREEPY branch follows one of the following semantics:

a) Reg1 = R0 and Reg3 = R0 and Link = 0: An unconditional branch that always jumps to the address in Reg2.
b) Link = 0: A conditional branch that jumps to the address in Reg2 (base) + Reg3 (offset) if Reg1 is 0. This is otherwise known as a *beqz* instruction.
c) Link = 1: A branch and link instruction to enable sub-routine calls and returns. The default next PC is stored into the link register and the program jumps to the address in Reg2.

4) Load/Store

Opcode	Reg1	Reg2	Reg3	Reserved
6b	5b	5b	5b	11b

Reg3 is the destination for a load and also is the source register for a store. The source/destination address for a load/store is given by Reg1 (base) + Reg2 (offset). Note that the memory address is hereby stored in an RRNS format.
Helper instructions such as *mov* etc. also exist, but are omitted from this description for brevity.

B. Error Model

First, we distinguish fault, error and failure as follows:
Fault: A single bit flips, but is not stuck-at, *i.e.,* only intermittent / transient faults are considered. Causes may range

from unreliable devices to low supply voltage to particle strikes to random noise and any combination therein.

Error: One or more faults in a single residue that show up during a consistency check.

Failure: The system has at least one error that it cannot detect, or has detected and cannot correct.

Faults may lead to errors which may lead to failures. We can guarantee the system is reliable if at most one error per core occurs between two consistency checks.

Redundancy in time, *i.e.*, check at cycle x, check again at cycle y, check again at cycle z, and vote, does not apply to this model as it is possible that the three checks suffer 3 independent 1 bit faults, rendering voting useless. The transient clause in the model rules out stuck-at faults. An implication of this is that we cannot achieve reliability by merely trading performance alone. Additional resources in terms of spatial redundancy are necessary, which is exactly what has been designed.

Different components of the core are protected via specialized means that target each component. The guiding principle is to design a system that uses the more efficient of RRNS/ECC based redundancy based on the range and nature of data being protected. Where both techniques are deemed insufficient to prevent the fault from metastasizing into an error, and eventually into a failure, the more conventional (and expensive) method: Triple Modular Redundancy (TMR), is employed. An alternative is to prevent the fault from occurring in the first place by using high V_{dd} (and/or circuit hardening). Choosing optimally between the latter expensive techniques is beyond the scope of this paper, but we assume that control signals' integrity is ensured using either TMR or intelligent state assignment, and that the RIU uses a high V_{dd} / hardened circuitry.

C. Signed Numbers Representation

In this section, we describe three competing ways of implicitly representing signed numbers, the first two of which were proposed by Waston [12], which we term Complement $M \times MR$ and Complement M representations, where, M is the product of all the non-redundant moduli ($M = m1 \times m2 \times m3 \times m4$) and MR is the product of all the redundant moduli ($MR = m5 \times m6$). To make up for the fact that Complement $M \times MR$ breaks the error correction algorithms and that Complement M is generally poor in performance, we propose a third approach, which we refer to as Excess-$\frac{M}{2}$ representation.

1) **Complement $M \times MR$ Signed Representation**
The $M \times MR$ complement signed representation is depicted by Figure 3. To provide a few examples, 0 is represented by 0, 1 is represented by 1, $\frac{M}{2} - 1$ is represented by $\frac{M}{2} - 1$, -1 is represented by $M \times MR - 1$ and $-\frac{M}{2}$ is represented by $M \times MR - \frac{M}{2}$. As can be seen, this is similar to signed binary representation. However, the known error correction algorithms break if numbers are represented in this manner.

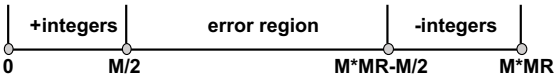


Fig. 3: Complement $M \times MR$ signed representation

2) **Complement M Signed Representation**
The complement M signed representation is depicted in Figure 4. This is similar to the $M \times MR$ method, except that the wrap-around occurs at M as opposed to $M \times MR$. This representation does not break error correction algorithms, provided that some correction factors (scaling and offset) are applied to the result of each arithmetic operation. However, further analysis indicates that calculating these correction factors [1] require knowledge of the signs of the operands, which is not explicitly known and sign determination in RRNS is a time-consuming process.

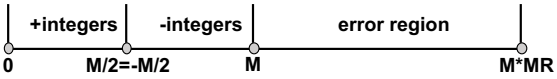


Fig. 4: Complement M signed representation

3) **Excess-$\frac{M}{2}$ Signed Representation**
The Excess-$\frac{M}{2}$ signed representation is shown in Figure 5. The excess notation, sometimes known as offset notation, merely shifts each number by $\frac{M}{2}$. To further elaborate, 0 is represented by $\frac{M}{2}$, 1 is represented by $\frac{M}{2} + 1$ and -1 is represented by $\frac{M}{2} - 1$. Similar to the M Complement representation, the results of arithmetic operations must be offset by a correction factor before they can be corrected. However, these correction factors turn out to be independent of the sign of the operands. So this method has great potential to improve the performance.
From the simulation results in V-D, we choose Excess-$\frac{M}{2}$ to be the defacto signed representation scheme for CREEPY.

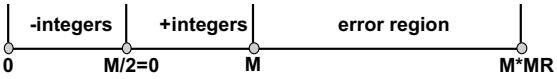


Fig. 5: Excess-$\frac{M}{2}$ signed representation

D. Residue Interaction Unit (RIU) Algorithms

The algorithm for single error correction was originally given by Watson [12]. However, RNS renders arithmetic overflow detection to be a non-trivial exercise. Furthermore, published work lacks sufficient details on the workings of overflow detection. In addition to providing a high level overview of the error correction algorithm, this section also presents algorithms for overflow detection. Furthermore, since the proposed algorithms augment the consistency checking algorithm itself, no extra hardware is warranted beyond that required by the error check.

[1]Correction factors are necessary to transform the results of an arithmetic operation before they can be checked for consistency. Details have been omitted due to space constraints.

1) Single Error Detection and Correction Algorithm: The single error detection and correction algorithm proposed by Watson [12] is based on an error correction table. The working steps of this algorithm for a system with 4 non-redundant moduli (m_1, m_2, m_3, m_4) and 2 redundant moduli (m_5, m_6), for any given integer X ($X < M = m_1m_2m_3m_4$) is as follows: (a) Use a base-extension algorithm to compute $|X'|_{m_5}$ and $|X'|_{m_6}$, where $|X|_m$=X mod m. (b) For $i = 5, 6$: compute $\Delta m_i = |X'|_{m_i} - |X|_{m_i}$. (c) A non-zero difference indicates the presence of an error. This pair of differences indexes into an entry of a pre-computed (fixed) error correction table, which contains the index of the residue that is in error and a correction offset that needs to be added to that residue to correct said error. In CREEPY, the error checking may be delayed. In other words, error may be stored in a register and fixed later.

TABLE II: Error Correction table of RRNS System with Moduli (3,5,2,7,11,13)

Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ	Δm_5,Δm_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	10 , 3	4 1
4 , 7	2 1	7 , 6	4 2		

For ease of presentation, we present such an error correction table for a smaller (toy) set of RRNS base moduli in Table II. The total entries in such a table is at most $2\sum_{i=1}^{4}(m_i - 1)$. For the reminder of this section, these set of bases are used for explanatory purposes.

2) Unsigned Number Overflow Detection: In the absence of any error or overflow, adding 2 unsigned RRNS numbers results in (Δm_5,Δm_6) = (0,0). In the absence of error, we observe that any overflow manifests itself as a fixed index into the error correction table, with the entry not corresponding to any error. Table III provides some examples of this observation. While computations of the deltas are most efficient by using a base-extension algorithm, we use the Chinese Remainder Theorem (CRT) or the Mixed-Radix Conversion (MRC) method here to first convert the RRNS number to binary, before computing deltas. This is solely for explanatory purposes; binary conversion is not actually necessary to detect overflow.

Iterating through all possible combinations of numbers and operations, we observe that the value pair of (Δm_5, Δm_6) is fixed. Moreover, (Δm_5, Δm_6) = (10,11) is not a legitimate address of the error correction table (Table II), thus enabling a distinction between an error and an overflow. This approach, however, does not apply to multiplication.

3) Signed Number Overflow Detection: Recall from Section IV-C that CREEPY uses the Excess-$\frac{M}{2}$ signed representation. We discuss the two sources of overflow independently:

1) *Add two positive numbers.* Table IV provides a few examples illustrating the algorithm. The $1 + 104$ in the first column is represented in decimal. After Excess-$\frac{M}{2}$ mapping, the computing equation is transformed to $106 + 209$ since $\frac{M}{2} = 105$ for the toy set of moduli. Therefore, the X RRNS value is the the RRNS of 106 and Y RRNS value is the the RRNS of 209. We observe that the pair (Δm_5,Δm_6) remains at a fixed value (10,11).
2) *Add two negative numbers.* Similarly, examples for adding two negative numbers are shown in Table V. In this case, we observe that the pair (Δm_5,Δm_6) is fixed to (1,2).

Note that neither (10, 11) nor (1, 2) are legitimate addresses in Table II, thereby enabling a distinction between an error and an overflow. However, while this method works for both addition and subtraction, it does not hold for detection of multiplication overflow as the delta-pair is not constant and sometimes indexes into a legal error correction table entry.

Figure 6 shows the overview of the whole algorithm.

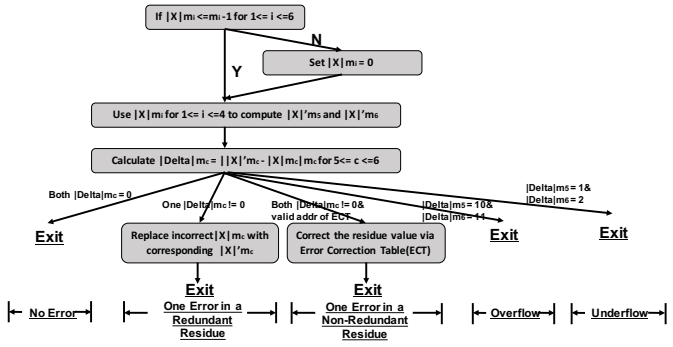


Fig. 6: Single error detection and correction algorithm with overflow/underflow detection

We observe that the described algorithm works in a similar manner even with our original set of bases, (199,233,194,239,251,509). An overflow results in a delta-pair of (77, 289), whereas an underflow results in (174, 220). Both these pairs do not index into legitimate entries of the error correction table for these set of bases (*cf.* Appendix E, Watson [12]).

V. Simulation

To measure the performance *vs* reliability trade-off of CREEPY, we augment a stochastic fault injection mechanism into a cycle-accurate, in-order, trace-based timing simulator. We abstract the notion of using marginal devices and/or near threshold voltage into E_{signal} and P_e. E_{signal}, provided as an input to the simulation, is a measure of the signal energy at the input of a gate; P_e is the probability of a fault occurring at the output of a gate in any given cycle. The relationship of E_{signal} and P_e can be defined by the following relation: $P_e = \exp(\frac{-E_{signal}}{kT})$.

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

TABLE III: Unsigned Number Overflow Examples in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	CRT/MRC	$\|X'\|_{m_5}$,$\|X'\|_{m_6}$	Δm_5, Δm_6
2+209	(2,2,0,2,2,2)	(2,4,1,6,0,1)	(1,1,1,1,2,3)	$(1,1,1,1) \Leftrightarrow 1$	$\|1\|_{11}$=1,$\|1\|_{13}$=1	10 11
3+209	(0,3,1,3,3,3)	(2,4,1,6,0,1)	(2,2,0,2,3,4)	$(2,2,0,2) \Leftrightarrow 2$	$\|2\|_{11}$=2,$\|2\|_{13}$=2	10 11
...	...	...	...	...	...	10 11
209+209	(2,4,1,6,0,1)	(2,4,1,6,0,1)	(1,3,0,5,0,2)	$(1,3,0,5) \Leftrightarrow 208$	$\|208\|_{11}$=10,$\|208\|_{13}$=0	10 11

TABLE IV: Excess-$\frac{M}{2}$ Overflow Examples for addition of two positive numbers in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	Add Correction Factors	CRT/MRC	$\|X'\|_{m_5}$,$\|X'\|_{m_6}$	Δm_5,Δm_6
1+104	(1,1,0,1,7,2)	(2,4,1,6,0,1)	(0,0,1,0,7,3)	(0,0,0,0,1,2)	$(0,0,0,0) \Leftrightarrow 0$	$\|0\|_{11}$=0,$\|0\|_{13}$=0	10 11
2+104	(2,2,1,2,8,3)	(2,4,1,6,0,1)	(1,1,0,1,8,4)	(1,1,1,1,2,3)	$(1,1,1,1) \Leftrightarrow 1$	$\|1\|_{11}$=1,$\|1\|_{13}$=1	10 11
...	...	...	...	...	...	...	10 11

TABLE V: Excess-$\frac{M}{2}$ Overflow Examples for addition of two negative numbers in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	Add Correction Factors	CRT/MRC	$\|X'\|_{m_5}$,$\|X'\|_{m_6}$	Δm_5,Δm_6
-1-105	(2,4,0,6,5,0)	(0,0,0,0,0,0)	(2,4,0,6,5,0)	(2,4,1,6,10,12)	$(2,4,1,6) \Leftrightarrow 209$	$\|209\|_{11}$=0,$\|209\|_{13}$=1	1 2
-3-104	(0,2,0,4,3,11)	(1,1,1,1,1,1)	(1,3,1,5,4,12)	(1,3,0,5,9,11)	$(1,3,0,5) \Leftrightarrow 208$	$\|208\|_{11}$=10,$\|208\|_{13}$=0	1 2
...	...	...	...	...	...	...	1 2

P_e: Probability of a fault occurring at the output of a gate in any given cycle, as already defined.

P_{add}: Probability of at least a single error in an adder (each sub-core has an adder). If there are N_{add} gates in an adder, the probability of each of these gates being free of error is $(1-P_e)^{N_{add}}$. Therefore, P_{add}=1-$(1-P_e)^{N_{add}}$. Similarly, P_{mul} is calculated. For multi-cycle operations, this definition holds as long as the output state of each gate is used exactly once for the operation, which is true for all of our operators.

P_{R_i}: Probability of at least 1 fault being present in a slice (sub-core) of register R_i between cycles t_1 and t_2, where, t_2 is the time at which the RRNS consistency of R_i is being checked and t_1 is the time at which the RRNS consistency of R_i was last established. As t_1 depends upon f_n (the check frequency, which we discuss later) and the dynamic instruction trace, we explicitly maintain a mapping of $LastCheckedCycle[R_i]$ in our simulator. Assuming an SRAM implementation of 8-bit wide R_i, the number of transistors is $8\times6=48$. The probability of R_i being error free for the entire duration of (t_1, t_2) is $(1-P')^{48(t_2-t_1)}$, where P' 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 $P'=100P\_e$. Therefore, $P_{R_i}=1-(1-100P_e)^{48(t_2-t_1)}$.

P_{loadX}: Probability of at least 1 fault being present in the loaded value of address X. This is clearly analogous to P_{R_i}, except that we maintain a mapping of $LastStoredCycle[X]$ to determine the last time a consistent state at address X was ensured. However, P_{loadX} also 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.

P_{SC}: Probability of at least 1 fault occurring in a sub-core from the last time it was checked. To illustrate, say an RRNS check was placed after the instruction $ADD\ R_3, R_2, R_1$. Then, $P_{SC}=1-(1-P_{add})(1-P_{R_2})(1-P_{R_1})$.

P_C: Probability of exactly 1 error occurring in a CREEPY core. This translates 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, $P_C=C_6^1\times P_{SC}(1-P_{SC})^5$, where the combinatorial choose operator C_n^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 CREEPY core from the last time it was checked. $P_C^0=6C_0\times(1-P_{SC})^6$.

P_C^{fail}: Probability of a CREEPY core failing. The current version of the CREEPY micro-architecture is unable to correct more than 1 error occurring in the core and defers recovery to a software checkpoint. As such, we deem ≥ 2 errors in the core as amounting to a failure. Therefore, $P_C^{fail}=\sum_{2\leq r\leq 6}C_6^r\times P_{SC}^r(1-P_{SC})^{6-r}=1-P_C^0-P_C$.

We statically compile integer benchmarks from the SPEC 2006 suite to generate a dynamic instruction trace compatible with the CREEPY ISA. The estimated gate count number for each of the five 8-bit subcores (not including the D Cache) is 2036, and, 2353 for the 9-bit one. Upon feeding this trace to the simulator, after every n^{th} instruction, as governed by f_n, the check frequency, an RRNS check is simulated. Based on P_C and P_C^0, it is stochasticaly determined if an RRNS correction must also be simulated. In accordance with the algorithms described in Section IV-D), we designate 8 cycles for the RRNS check and an additional 1 cycle should an error be corrected. At the end of each RRNS check, $P_{C,i}^{fail}$ is used to determine if a failure is likely to occur at that cycle t_i. As such, we use a typically used reliability metric, Mean Time To Failure (MTTF), which can be defined as follows: $MTTF=\frac{Total\ Cycles}{CPU\ \ Frequency\ \times\ \sum_i P_{C,i}^{fail}}$.

In addition to varying E_{signal}, we explore following optimizations that are expected to improve reliability and performance:

a) : Vary check frequency f_n, where f_n denotes that every n^{th} instruction is checked for an RRNS error. Intuitively, checking very frequently (ex. f_1; checking every instruction) favors higher reliability, whereas checking very infrequently (ex. f_∞, or equivalently, f_0) favors higher performance.

b) : For $n > 1$, perform a pipeline check on up to n destination registers / memory locations. This check strategy, known as $pipe_n$ is similar to f_n in that the consistency check action only happens after the n^{th} instruction, but with the added action that checks all the output destinations for the past n instructions in a pipeline fashion. Assuming that the original check takes 8 cycles, the check of $pipe_n$ should be 8+(n-1) cycles.

c) : For $n > 1$, in addition to performing a check (either f_n or $pipe_n$) every n^{th} instruction, also perform a check at every store instruction to enhance memory (cache) reliability. For brevity, we turn on this optimization for all the results presented in this section.

A. Performance vs Consistency Check Frequencies

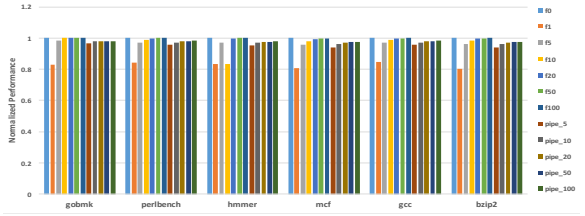


Fig. 7: Performances VS Consistency Check Frequencies

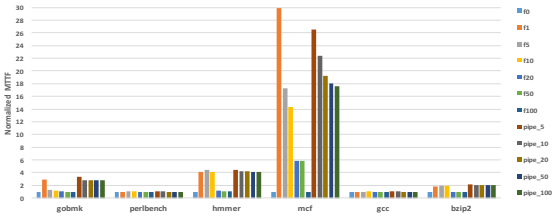


Fig. 8: MTTFs VS Consistency Check Frequencies

Figure 7 shows the relationship between system performance and consistency check frequency. We vary the check frequency from 0 to 100 in both non-pipeline and pipeline approaches. f_0 means that no extra consistency checks are performed whatsoever, and f_{100} performs an extra consistency check after every 100th instruction. Also recall that $pipe_5$ indicates that the results of each instruction are checked at the end of every 5^{th} instruction. The Y-axis is normalized against the baseline, which is f_0. The results show that the data trends of all the benchmarks are very similar. f_0 always gets the best performance because it incurs no consistency check overhead. Frequently checking for errors hurts performance, as evidenced by f_1 suffering from the worst performance degradation. Pipelined checks are very close to non-pipelined checks in performance on average. In other words, using the pipelined check approach enables a potentially more reliable system without sacrificing performance when compared to the non-pipelined check approach.

B. MTTF vs Consistency Check Frequencies

As explained earlier, Mean Time To Failure (MTTF) provides a measure of reliability. Similar to Section V-A, we normalize the MTTF against the baseline, f_0. Intuitively, a higher check frequency would result in a higher MTTF as the probability of an uncorrected error diminishes. However, it can be seen from Figure 8 that f_1 does not always provide the highest MTTF. This can be attributed to the fact that the consistency checks themselves are not instantaneous and this added delay leaves gates more vulnerable to faults.

For example, consider the f_1 and f_{10} scenarios below. All add instructions take 1 cycle and check instructions take 8 cycles. In instruction 0, R_5 will be checked in both cases. Then after this step, R_5 is first used in instruction 11. From the evaluation model we defined, the fault probability of R_5 depends on the time intervals between instruction 0 and 11 (time interval from last check). The time interval for the f_1 scenario in this example is $(1 + 8) \times 10 = 90$ cycles, but for f_{10}, this is only $1 \times 10 + 8 = 18$ cycles. Therefore, the fault probability of R_5 in instruction 11, is higher for the f_1 scenario than for f_{10}. Therefore, a low frequency pipeline checking (such as $pipe_5$ or $pipe_{10}$) achieves a good balance between performance and reliability.

f_1:
(0) add R_5, R_2, R_3
check R_5
(1) add R_1, R_2, R_3
check R_1
(2) add R_3, R_2, R_1
check R_3
...../no check R_5
(10) add R_8, R_2, R_1
check R_8
(11) add R_{11}, R_3, R_5

f_{10}:
(0) add R_5, R_2, R_3
check R_5
(1) add R_1, R_2, R_3
(2) add R_3, R_2, R_1
.....
(10) add R_8, R_2, R_1
check R_8
(11) add R_{11}, R_3, R_5

C. MTTF vs Energy Input Per Gate

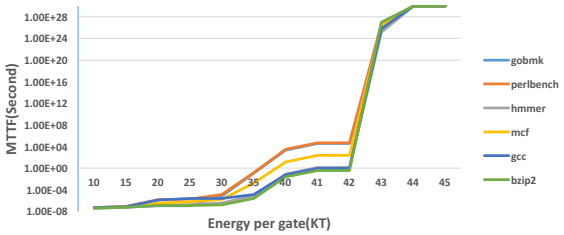


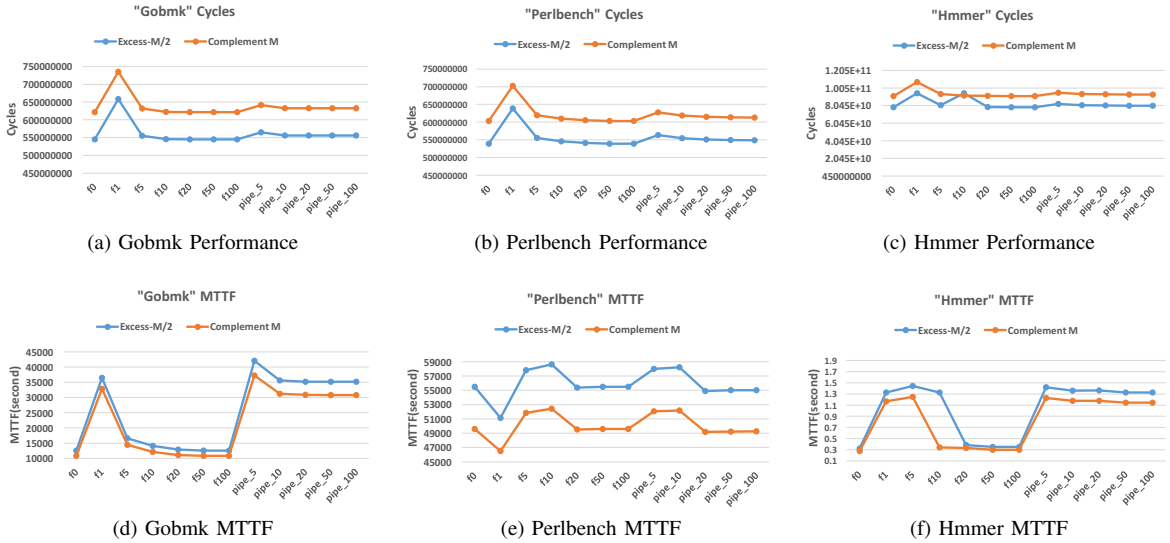
Fig. 10: MTTFs VS Energy input per gate

Figure 10 plots the simulated MTTF for various values of E_{signal}. For brevity, only the f_1 paradigm is depicted here. We notice that a value of E_{signal} greater than 44kT results in infinite MTTF, which is essentially a very large number, given the precision limits of the simulation. However, we also observe that the MTTFs drop very fast when the E_{signal} is between 42kT and 43kT.

The key take away from Figures 7, 8 and 10 is that a lower E_{signal} can still lead to acceptable latency without degradation in output quality.

D. Excess-$\frac{M}{2}$ vs Complement M signed representation

There is scope for further tuning the CREEPY core by employing alternate representations and algorithms. For instance,



(a) Gobmk Performance (b) Perlbench Performance (c) Hmmer Performance

(d) Gobmk MTTF (e) Perlbench MTTF (f) Hmmer MTTF

Fig. 9: Performance and MTTF Comparisons for signed number representation methods: "Excess-$\frac{M}{2}$" and "Complement M"

there may be a set of bases that are conducive to very fast consistency checks. In this section, we compare two of the signed representations that were discussed in Section IV-C. The performance and reliability comparisons are shown in Figure 9. Recall that consistent arithmetic operations using the Complement M method require the need to determine the sign of operands, which is a time-consuming operation in RRNS. Therefore, the performance of Excess-$\frac{M}{2}$ is much better than Complement M on average. Even for MTTF, Excess-$\frac{M}{2}$ is better than Complement M. The reason is similar to that described in Section V-B (larger time intervals between consistency checks imply higher probability for system failure).

VI. Related work and Conclusion

While the concepts of RNS, RRNS, error correction, device limits and signal integrity by themselves are not new, we believe this is the first proposal that relates these mathematical and physical entities to push back the horizon of Moore's law for high-throughput exascale HPC systems. It must be noted that our approach does not require the algorithm to be designed in a fault tolerant manner, thereby expanding the scope of CREEPY to Turing completeness.

Prior work in the Digital Signal Processors (DSP) domain ([4], [5], [11]) has focused on RNS datapaths to take advantage of fine-grained data parallelism and energy efficient properties that RNS operations provide. By utilizing RRNS, CREEPY benefits from these, but improves upon generality and energy-efficiency. Chiang et al. [3] provide RNS algorithms for comparison and overflow detection, but assume all bases to be odd and do not consider error correction.

CREEPY, being an RRNS computer, draws its underlying mathematics heavily from the pioneering work of Watson and Hastings [8], [12], [13]. However, at the time, they probably did not deem it necessary to provide a detailed micro-architecture and ISA to support their algorithms. CREEPY extends and improvizes on their RRNS algorithms in addition to providing a detailed design. This paper describes and demonstrates the usability of a CREEPY computer that improves energy efficiency by adding computationally redundant hardware.

References

[1] D. E. Nikonov and I. A. Young, "Overview of Beyond-CMOS Devices and a Uniform Methodology for Their Benchmarking," in Proceedings of the IEEE, vol. 101, no. 12, pp. 2498-2533, Dec. 2013.

[2] McMenamin Adrian, *The end of Dennard scaling*, 2013.

[3] J.-S. Chiang and M. Lu, *Floating-point numbers in residue number systems*, Computers and Mathematics with Applications,22, no. 10, 127-140, 1991.

[4] Rooju Chokshi , Krzysztof S. Berezowski , Aviral Shrivastava , Stanislaw J. Piestrak, *Exploiting residue number system for power-efficient digital signal processing in embedded processors*, Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems, October 11-16, 2009, Grenoble, France

[5] E. Di Claudio, F. Piazza and G. Orlandi, *Fast Combinatorial RNS Processors for DSP Applications*, IEEE Trans. Computers, vol. 44, no. 5, pp. 624-633, May 1995.

[6] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, *Design of ion-implanted mosfets with very small physical dimensions*, IEEE Journal of Solid-State Circuits, 9, October 1974.

[7] H. L. Garner, *The Residue Number System*, IRE Transactions on Electronic Computers, pp. 140-147, June 1959.

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

[9] Lyons, Robert E., and Wouter Vanderkulk. "The use of triple-modular redundancy to improve computer reliability." IBM Journal of Research and Development 6.2 (1962): 200-209.

[10] MacWilliams F J, Sloane N J A, *The theory of error correcting codes[M]*, Elsevier, 1977.

[11] J. Ramirez, *RNS-enabled digital signal processor design*, Electron. Lett., vol. 38, no. 6, pp. 266-268, 2002

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

[13] R. W. Watson and C. W. Hastings, *Self-checked computation using residue arithmetic*, Proc. IEEE, vol. 54, pp. 1920-1931, Dec. 1966.