2018 IEEE International Symposium on High Performance Computer Architecture

Memory System Design for Ultra Low Power, Computationally Error Resilient Processor Microarchitectures

Sriseshan Srikanth*, Paul G. Rabbat†, Eric R. Hein*, Bobin Deng*, Thomas M. Conte*,
Erik DeBenedictis‡, Jeanine Cook‡ and Michael P. Frank‡
*Georgia Institute of Technology. Email: {seshan}{ehein6}{bdeng}{tom}@gatech.edu
†Intel Corporation. Email: paul.g.rabbat@intel.com
‡Sandia National Laboratories. Email: {epdeben}{jeacook}{mpfrank}@sandia.gov

***Abstract*—Dennard scaling ended a decade ago. Energy reduction by lowering supply voltage has been limited because of guard bands and a subthreshold slope of over 60mV/decade in MOSFETs. On the other hand, newly-proposed logic devices maintain a high on/off ratio for drain currents even at significantly lower operating voltages. However, such ultra low power technology would eventually suffer from intermittent errors in logic as a result of operating close to the thermal noise floor. Computational error correction mitigates this issue by efficiently correcting stochastic bit errors that may occur in computational logic operating at low signal energies, thereby allowing for energy reduction by lowering supply voltage to tens of millivolts.**

Cores based on a Redundant Residual Number System (RRNS), which represents a number using a tuple of smaller numbers, are a promising candidate for implementing energy-efficient computational error correction. However, prior RRNS core microarchitectures abstract away the memory hierarchy and do not consider the power-performance impact of RNS-based memory addressing. When compared with a non-error-correcting core addressing memory in binary, naive RNS-based memory addressing schemes cause a slowdown of over 3x/2x for inorder/out-of-order cores respectively. In this paper, we analyze RNS-based memory access pattern behavior and provide solutions in the form of novel schemes and the resulting design space exploration, thereby, extending and enabling a tangible, ultra low power RRNS based architecture.

I. Introduction

We hit the power wall in 2005, which meant that increasing clock frequency was no longer a technique to improve performance. Single core performance plateaued as a result of the power wall although there is more instruction level parallelism (ILP) inherent in programs, waiting to be exploited [11], [35], [58]. The community has since then adopted multiple cores and specialized accelerators to make better use of Moore's law, albeit at the cost of programmability. The phenomenon of transistors retaining their power density as they scaled down (known as Dennard Scaling [22]) ended a decade ago [69], meaning that adding more and more transistors no longer improved performance without also significantly increasing energy consumption. A sure-shot mitigation technique is to strive to improve the fundamental single core power-performance profile from the ground up.

Dynamic power scales proportionately with frequency and with the square of operating voltage, therefore, lowering V_{dd} is more beneficial. Although the theoretical lower limit for V_{dd} for inverter functionality is $2\frac{kT}{q}$ (or about $36mV$) [49], [96], [110], there are challenges to operating at this level [13]. With conventional MOSFETs, reducing supply voltage below ˜0.7V results in increased switching delay, irrespective of channel length. Coupled with their gentle subthreshold slope ($> 60mV/decade$), the upshot is that V_{dd} reduction actually causes higher leakage energy, negating the benefits of the reduction in the first place and furthermore forcing a significantly slower clock rate.

Theis and Solomon [105] suggest that new device concepts [76] within the purview of two-dimensional lithography technology, such as tunneling FETs, enable reduction of the $\frac{1}{2}CV^2$ energy to small multiples of kT, without resulting in a significantly low switching speed [104] when compared to MOSFETs. Similarly, research on ferroelectric transistors, *aka* negative capacitance FETs (NCFETs) demonstrates a sub-$60mV/dec$ slope as well as a higher drive current [50]–[52], [87], both of which are necessary in rendering V_{dd} reduction beneficial to energy reduction without significantly sacrificing performance, when compared to MOSFETs.

These next generation devices are fast switching even at few tens of millivolts, but as a result, are vulnerable to thermal noise perturbations. This translates into *intermittent, stochastic bit errors in logic*. With signal energies approaching the kT noise floor, future architectures will need to treat reliability as a first class citizen, by employing efficient computational error correction.

A. Computational Error Correction

Standard error correcting codes (ECC) [66] 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) [109], requiring over a 200% overhead in area and energy for single error correcting/double error detecting (SECDED) capability. Several techniques in the form of arithmetic codes such as AN codes [9], [28], [29], [62], [88], [111], self-checking [45], [48], [67], [73]–[75], [107] and self-correcting [25], [32], [41], [55], [68], [79], [83], [84], [95], [106] adders and multipliers have since been devised. Orthogonally, there have been proposals that employ redundancy at a higher granularity, such as timing speculation (wherein error correction capability is limited to circuit timing violations) [26], [37], partial pipeline replication [1] or checkpoint-rollback-recovery such as those in IBM POWER6/7/8 and z10/196 [8], [16], [44], [61] processors, and various Intel Corporation [90] and Sun Microsystems mainframes [43]. While these are more efficient than naive TMR, they come with limitations on their error model, their area overheads are still over 100% and/or they incur a significant performance penalty [91] (e.g., owing to the fact that they leverage temporal redundancy in an effort to minimize area overhead).

There exists a class of computationally error resilient codes based on the residue number system (RNS) [31], that

This work was approved for public release by Sandia National Laboratories SAND2017-12733.

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00065

IEEE computer society

Table I: A (4, 2)-RRNS example with the simplified base set (3, 5, 2, 7, 11, 13). % is the mod operator. Range is 210, with 11 and 13 being the redundant bases. (Reproduced from [21].)

Decimal	% 3	% 5	% 2	% 7	% 11	% 13
13	1	3	1	6	2	0
14	2	4	0	0	3	1
13+14=27	(1+2)% 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.					

are generally superior to the above techniques in terms of area, energy and latency overheads. The premise of RNS is that a number can be uniquely represented as a tuple of residues, where the residues are the remainder when the number is divided by a set of coprime bases/moduli. Each residue itself may be represented in a weighted radix representation in binary. An example is shown in Table I. Assume the set of bases/moduli to be (3, 5, 2, 7); then, the number 13 can be uniquely mapped to the tuple (1, 3, 1, 6) as a result of the Chinese remainder theorem. Observe that (13 mod 3 = 1), (13 mod 5 = 3) and so on. Now, suppose we wish to add 13 (1, 3, 1, 6) to the number 14 (2, 4, 0, 0); this can be achieved by simply (modulo) adding the residues respectively to obtain 27 (0, 2, 1, 6). Observe that ((1 + 2) mod 3 = 0), ((3 + 4) mod 5 = 2) and so on. Critically, the computation occurs with no carries or interaction between residues.

RNS is similarly closed under subtraction and multiplication operations as well, and they too can operate without interaction between residues. This important property has several useful implications:

- As the residues operate with no carries between them, they can operate in parallel. Furthermore, each residue operation's bit width is a fraction of that of the original operation. This translates into improved computational efficiency, which has been proven to be especially useful in digital signal processing (DSP) [19], [24], [81].
- A very large number can be losslessly represented and operated on via many smaller numbers (residues) in parallel. This property is used by the cryptography (RSA) community [3], [42], [116].
- Any bit error caused by a faulty computational logic element is guaranteed to be localized to within the corresponding residue, without impacting any of the other residues.

The last implication is of particular interest because it allows for a robust, efficient method for computational error correction. When redundant bases/moduli are introduced (Redundant RNS or RRNS) [112], the resulting redundant residues form an error correcting code that transforms itself automatically upon arithmetic operations, rather than having to be recomputed afterwards. If a corrupt residue results during an arithmetic operation, it is possible to infer its value using the remaining correct residues in the result. To continue our running example in Table I, the number 13, being less than the product of the initial set of bases/moduli (3, 5, 2, 7), (i.e, 210), it can be uniquely represented using the 4-tuple (1, 3, 1, 6), via the Chinese remainder theorem. As such, we refer to these bases/moduli and residues as non-redundant. Upon adding two redundant bases/moduli, say (11, 13), the resultant redundant residues are therefore (2, 0). These redundant residues, by definition, are not necessary for representation, but provide a way to recover from errors. Given the (4, 2)-tuples for 13 (1, 3, 1, 6, 2,0) and 14 (2, 4, 0, 0, 3,1), a storage/transmission/computation error arising in any one of the residues of 13, 14 or their arithmetic manipulation result, can be corrected using the remaining residues. The running example is summarized in Table I.

The range for an RRNS representation is the product of its non-redundant moduli. Therefore, by choosing a non-redundant base/modulus set to be (199, 233, 194, 239) it becomes possible to represent a 32-bit integer in general in an RNS format, and extending it with a redundant set of (251, 509) allows for an RRNS representation. Such a (4, 2)-RRNS has the following advantages, over and above what RNS provides to us:

- Computational error correction can be achieved with a little over 50% area overhead.
- The SECDED granularity is that of a residue; meaning that such an RRNS is capable of *correcting multi-bit errors as long as they occur within a single residue*, or alternately, is capable of *detecting multi-bit errors as long as they occur within at most 2 residues*.
- Being closed under arithmetic, the correct value is *preserved* across a chain of dependent operations. Therefore, it is not necessary to incur the overhead of an RRNS error correction/detection after every operation.
- Due to the above, RRNS lends us robust computational error correction with relatively insignificant performance penalty, as also evidenced by Deng et al. [21].

Furthermore, there has been a significant body of research on RRNS that strives to make it more efficient algorithmically for error resilience [5], [6], [17], [23], [27], [33], [34], [38], [47], [56], [57], [63], [77], [80], [82], [89], [92]–[94], [97], [98], [101]–[103], [113]–[115], [117] and division [4], [30], [39], [40], [42], [63], [64], [92], [99]. Although typically limited to detection, residue based logic protection (including that for floating point units and vectorized units) has widespread used in several commercial high-end server processors [8], [16], [43], [61], [90]. Clearly, the RRNS approach of computational error correction is ranked among the highest in terms of error correction capability and efficiency.

An efficient RRNS-based architecture is a viable compiler target for contemporary general-purpose as well as scientific computing applications, *in addition to being able to work efficiently with novel devices that operate close to the kT noise floor*. We strongly believe that single-thread processor performance scaling that has been stalled since the mid-2000s can be restarted via RRNS.

B. The Problem

While RRNS is clearly attractive for designing ultra-low-power, computationally error resilient microarchitectures, prior work on RRNS has abstracted away the memory hierarchy for simplicity. Thus, an RRNS microarchitecture hits a performance bottleneck when connected to non-ideal, real world memory systems. Memory is addressed in binary in conventional processors, whereas an RRNS compute core natively generates memory addresses as a tuple of residues. There are two naive approaches to this memory interface problem:

Binary. Convert the tuple-of-residues format to binary and address memory in binary as usual. This approach imposes a severe latency and energy penalty on every instruction fetch, load and store operation. This overhead is due to the multi-step nature of each RNS to binary conversion, which, nominally costs 8 cycles in the form of add and table lookup operations. According to our results, this slowdown is 3× on average for in-order cores and 2× for out-of-order (OoO) cores.

Rns_concat. Another straightforward, although naive approach is to concatenate the native tuple-of-residues and use the result as the memory address. Unfortunately, this technique destroys spatial locality of sequential memory accesses, rendering caches largely ineffective and causing application slowdowns of over 3× on average for in-order cores and 4× for OoO cores.

There are other overheads associated with RRNS, such as non-trivial comparison and boolean operations. However, the overheads due to memory addressing inefficiencies are significantly higher. The memory hierarchy is accessed for *each* instruction (PC) in addition to memory instructions (LD/ST). In contrast, comparison and boolean op instructions are less frequent, even if a targeted code generator does not minimize such ops. Furthermore, even with a naive implementation, the penalty for such ops is less than that of a cache miss. In an independent study (that ignores memory addressing inefficiencies), we found that the impact of slow comparison, boolean operations as well as consistency checking operations due to RRNS makes programs run just about 20% slower than a traditional core. Yet, due to RRNS, we realize significant energy savings, rendering energy-delay-product benefits of about 2× when compared to traditional non-error-correcting cores, in spite of these overheads.

In spite of its strong potential to restart single thread performance scaling, an RRNS processor is not competitive with traditional designs due to memory addressing inefficiencies outlined above. The energy savings would be overshadowed by the decrease in memory system performance. This paper is the first study of its kind to our knowledge to focus on the RRNS memory access problem.

C. Contributions

This paper makes the following contributions:

1) Extends an RRNS microarchitecture - one that is capable of reliably functioning with ultra-low energy logic devices that operate near the kT thermal noise floor at tens of millivolts - to support efficient memory hierarchy access.
2) Proposes and analyzes an efficient, novel translation scheme (*rns_sub*) with locality properties similar to that of *binary*, but with a fraction of its performance/energy overhead.
3) Reduces the performance impact of the naive *binary* approach by using a TLB-like structure called the *Conversion Lookaside Buffer* (CLB) to cache conversions.
4) Proposes a technique to improve spatial locality in the native, zero-overhead translation scheme (*rns_concat*) via a hybrid compiler approach and a modified programming model.
5) Constructs a design space from the schemes proposed and from the analysis of the resultant memory access pattern behavior, along with a detailed cost-benefit analysis.

II. RRNS Core Microarchitecture

Having already presented an overview of the general workings of RNS and RRNS, we now provide a formal description for completeness (proofs omitted for brevity). We then describe an overview of a generic RRNS compute core. Readers who are not interested in the formality can safely skip ahead to Section II-C.

A. Residue Number System (RNS)

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 bijectively 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 can be represented as the following tuple: $(|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, ..., |x|_{m_n})$, where $|x|_m = x \ mod \ m$. Each term in this n-tuple is referred to as a residue. If necessary the value of x can be regenerated from the tuple of residues using a series of addition and table lookup operations via the Chinese remainder theorem, mixed-radix conversion, or macro-coefficient extraction.

Addition, subtraction and multiplication are closed under RNS, rendering the residues to be mutually independent *wrt* arithmetic. In other words, 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.

B. Redundant RNS

To augment RNS with fault tolerance, r redundant bases are introduced. The set of moduli now contains n non-redundant and r redundant moduli: $B = \{m_i \in \mathbb{N} \text{ for } i = 1,2,3,...,n,n+1,...,n+r\}$. 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. Recall that the introduction of r redundant residues renders a computationally resilient *error code* because of the fact that all residues are transformed in an identical manner under arithmetic operations. For x such that $x \in \mathbb{N},\ x < M$, its RRNS representation is as follows: $(|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, ..., |x|_{m_n}, |x|_{m_{n+1}}, ..., |x|_{m_{n+r}})$, i.e., containing n non-redundant residues as well as r redundant residues.

We refer the interested reader to [112] for details of the multi-bit error correction operation, which involves addition, multiplication and table lookup operations. This correction capability increases with r, tolerating upto $\frac{r}{2}$ errant residues [33]. There are proposals to perform fractional multiplication [112] and to represent floating point numbers [18] using RNS. The key idea to extend this to RRNS is to protect the exponent and mantissa separately, as they transform differently upon arithmetic operations. A detailed treatment of these concepts is beyond the scope of this paper.

For fault tolerance to work, certain conditions must be satisfied by B [112]. Given these, published work suggests that the most efficient conversion to *binary* algorithm (for $n = 4$, independent of r) nominally costs 8 cycles and is possible via macro-coefficient extraction [112]. More efficient conversion algorithms for RNS exist, such as those that use Mersenne primes (or other specific structural properties) as bases [14], [15], [71], but no known RRNS algorithms exist.

In this paper, $(n,r) = (4,2)$ and moduli set used is (199, 233, 194, 239, 251, 509), whose non-redundant representable range is roughly that of a 32 bit unsigned integer. A different set of bases may also be used for reasons such as a higher representable range, more efficient arithmetic, improved reliability characteristics etc., the details of which are beyond the scope of this paper. We assume the aforementioned bases in our evaluation but are careful not to over-fit our architectural suggestions and insights to this specific set.

C. Anatomy of an RRNS Core Microarchitecture

Figure 1 depicts a generic schematic of an RRNS core with 4 non-redundant *sub*-cores and 2 redundant ones. This schematic is similar to prior RRNS core microarchitecture proposals [21], [112]. Each *sub*-core is associated with exactly one base modulus and thereby operates on its own

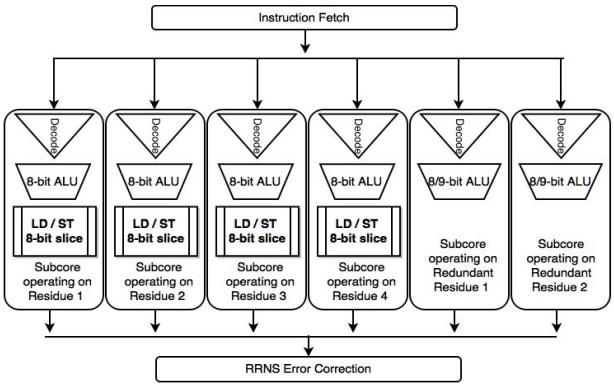


Figure 1: Generic high level schematic of an RRNS error correcting compute core with 4 non-redundant *sub*-cores and 2 redundant *sub*-cores.

residue, with the register file and highest level data cache being distributed into the 6 subcores on a per-residue basis.

Error model. Recall that such a core operates entirely on RRNS data and literals, meaning that there are no unnecessary (and expensive) conversions to and from *binary*. Therefore, all memory addresses (PC, LD/ST) are in RRNS form, including any pointer arithmetic. Another upshot of operating entirely on RRNS data is that control-path errors manifest themselves as data errors, meaning that they can be handled simply by handling the data error. For example, if there is an error in bypass logic in a subcore, or, if a faulty decoder in one of the subcores causes it to perform a multiplication instead of an addition, the resultant residue for that subcore would have an erroneous value, but can be recovered from the remaining 5 residues that were a result of the correct addition operation. Finally, as instructions themselves don't undergo modification, ECC is sufficient to protect them [21]. The architecture proposed in [21] is able to ensure reliable operation as long as at most one subcore is in error (possibly multi-bit) between two error correction operations. Circuit-hardening or using high V_{dd} for the error correction logic is proposed to ensure its reliable operation. Because of latching effects, SRAM transistors have different (more prone to errors) reliability characteristics when compared to logic, and as such, are modeled with 100× the error probability of logic transistors.

Overheads. In terms of area, such an RRNS error correcting core requires 2× the area of a traditional non-error-resilient core. However, a large fraction of this overhead is from LUTs necessary in the error correction operation; a (4, 1)-RRNS core that simply detects errors is in fact 34% smaller in area when compared to a traditional core. In an independent study that implements the error model above (but ignores memory addressing inefficiencies), we found that the overhead due to slow comparison operations, boolean operations as well as RRNS error correction operations resulted in an overhead of just about 20% in runtime when compared to a traditional, non-error-correcting core over general purpose (SPEC2006) as well as computationally intensive workloads (such as FFT, matrix multiplication etc.). However, RRNS enables lowering of signal energy to few tens of millivolts, and results in about a 2× improvement in energy-delay-product, in spite of these overheads. The engineering details of this study are beyond the scope of this paper and we omit them for space constraints.

Abstractions for the memory interface. The presence of a Load/Store unit in the redundant *sub*-cores is implementation dependent. For the purposes of this paper, we assume that the core is responsible for generating *valid* memory requests in the Memory Address Register (MAR). This is possible by either (a) inserting an RRNS consistency check before a memory access, or (b) by enabling a checkpointing mechanism for rollback/recovery in case a memory request to an illegal location is generated. For the latter, a segmentation scheme can be used to flag "wrongly" computed addresses as illegal, thereby triggering checkpoint recovery. As we show in Section III-B, minor perturbations in the native *rns_concat* representation of a memory address causes its value to fluctuate wildly, thereby allowing for such segmentation to work. Note that the consistency check of (a) or the segment check of (b) can be done in parallel with a memory access, and are therefore not on the critical path of the program. Therefore, we omit the Load/Store units in the redundant subcores. Finally, the addressing logic in the memory hierarchy (such as a decoder or an address translator) is assumed to either utilize devices that do not stochastically flip due to thermal noise, or employ self checking logic [36], [86]. Relaxing either assumption reveals a set of implementation dependent tradeoffs and are left for future work. *Without loss of generality, we assume for the purposes of this paper that no explicit error correction is required for such addressing logic in the memory hierarchy*. It follows that the redundant residues can be dropped from the MAR, and that a 32-bit address can be logically represented as a 4-tuple RNS number.

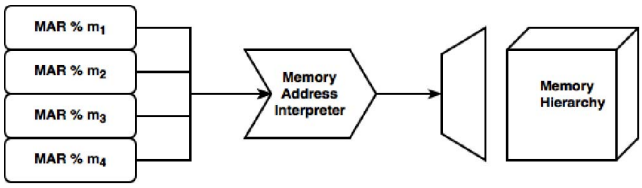


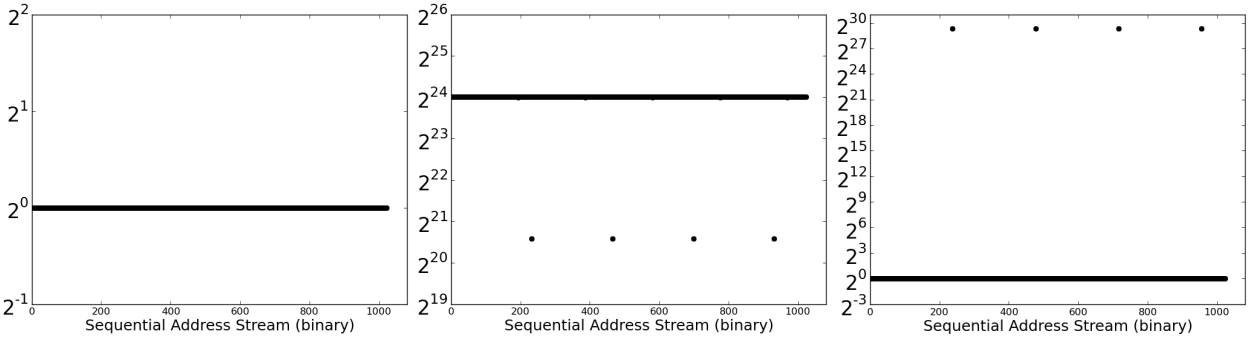
Figure 2: An (R)RNS compute core natively generates memory addresses in the 4-residue tuple format. This is depicted by MAR % m_i in the figure, where m_i is the i^{th} base modulus, % is the modulo operator, and MAR could be one of Program Counter (PC), Load or Store address.

III. Memory Addressing Schemes

As noted in Section II-C, the memory address generated by an RRNS compute core in its native form is in a tuple-of-residues format, where the address itself may be to instructions or data. As depicted in Figure 2, such a 4-tuple must be properly interpreted before the memory hierarchy can be accessed. In this section, we present a basis set of possible *interpretations* of an address.

A. Interpretation Schemes

Binary. The approach assumed by prior work was to convert the 4-tuple of residues to its binary representation and use this as a memory address. From this point on, the system can access the memory hierarchy as conventional computers do. However, this conversion from RNS to *binary* doesn't come for free, and the cost must be paid for each memory access, both cache hits and cache misses. This overhead is a multi-cycle access time increase and an associated increase in energy consumption. This is because conversion from RNS to *binary* is non-trivial: it is a multi-step operation, involving addition and table-lookup operations, and costs 8 CPU cycles nominally. Our experiments indicate that applications suffer from a slowdown of 3× on average for inorder cores and 2× for OoO cores upon using such a naive conversion approach.



(a) Delta between consecutive addresses: *binary* scheme.

(b) Delta between consecutive addresses: *rns_concat* scheme.

(c) Delta between consecutive addresses: *rns_sub* scheme.

Figure 3: Spatial locality analysis via visual comparison of the 3 addressing schemes for a sequential stream of addresses by computing the *deltas* of *interpreted* addresses of consecutive addresses of the sequential stream.

$$\{r_1, r_2, r_3, r_4\} \implies binary(\{r_1, r_2, r_3, r_4\})$$

This conversion typically is not a one-time cost as addresses may repeat themselves (locality). As a solution, we propose to cache the conversion itself in a Conversion Lookaside Buffer (**CLB**), in a manner similar to how a TLB caches translations.

Rns_concat. On the other extreme, we have a lightweight interpreter that merely concatenates the 4-tuple, with the resulting 32bit number being treated as the address to the memory hierarchy. More concretely, the 4-residue tuple $\{r_1, r_2, r_3, r_4\}$ is treated as the bitstream $r_1 r_2 r_3 r_4$.

$$\{r_1, r_2, r_3, r_4\} \implies r_1 r_2 r_3 r_4$$

Rns_sub. We define another scheme similar to *rns_concat*, but the lowest residue is subtracted from the 3 other residues in an attempt to preserve locality.

$$\{r_1, r_2, r_3, r_4\} \implies rns\_concat((r_1 - r_4)\%m_1, \\ (r_2 - r_4)\%m_2,\ (r_3 - r_4)\%m_3,\ r_4)$$

where, the RNS base moduli are given by m_i. This scheme is significantly less expensive than *binary* and is slightly more expensive than *rns_concat*.

B. Sequential Address Analysis Example

Figure 3 shows a comparison of how the 3 addressing schemes discussed so far, *binary*, *rns_concat* and *rns_sub*, remap a stream of sequential accesses into the memory address space. On the X-axis is the input value to the Memory Address Interpreter; on the Y-axis is the difference (*delta*) between two consecutive *interpreted* memory addresses.

First, notice that the *delta* for *binary* is always exactly 1; converting the tuple-of-residues back to the binary number they represent results in sequential memory addresses. However, *rns_concat* remaps accesses according to the following function: if the address $X \equiv \{r_1,\ r_2,\ r_3,\ r_4\}$, then $X+1 \equiv \{r_1+1,\ r_2+1,\ r_3+1,\ r_4+1\}$, thereby resulting in a *delta* of 0x01010101 as each residue is 8-bits wide. This is the *delta* value that is seen in the figure as a straight horizontal line slightly above 2^{24}. However, each time a residue overflows, the above constancy claim for *delta* breaks down, generating the discontinuities shown in the plot.

Non-unit delta values will cause consecutive accesses to touch *different* cache lines and memory pages, destroying spatial locality in the access stream. To mitigate this constant *delta* offset seen in *rns_concat*, we define *rns_sub* in an effort to preserve locality. Applying *rns_sub* to X and 1, we observe the following pattern (ignoring modulo overflows):

$$X \equiv \{r_1 - r_4, r_2 - r_4, r_3 - r_4, r_4\};\quad 1 \equiv \{0,0,0,1\}$$
$$\implies X+1 \equiv \{r_1 - r_4, r_2 - r_4, r_3 - r_4, r_4 + 1\}$$

Therefore, *rns_sub* yields a *delta* value of exactly 1 in the common case, similar to the *binary* scenario. In general, the *rns_sub* representation of any number $n < m_4$ is $\{0,0,0,n\}$, meaning that difference between $X+n$ and X in the *rns_sub* representation is exactly n in the common case. This means that *rns_sub* is capable of preserving the spatial locality that cache and DRAM memory systems need to deliver performance.

While the detailed evaluation of a prefetcher is beyond the scope of this paper, intuitively, prefetchers that work well with *binary* would work well with *rns_sub* as well. With *rns_concat*, however, a stride of 1 must be re-interpreted as a stride of 0x01010101, which is the *delta* between consecutive addresses, significantly altering the operation of most prefetchers.

Although our example analysis in this section is based upon a sequential address stream, the insights are applicable to arbitrary streams that exhibit spatial locality, as is demonstrated via simulation results detailed below.

IV. Design Tradeoffs

Utilizing a basis set of address interpretation schemes outlined in Section III, we now construct and analyze a design space consisting of memory access granularity, Memory Address Interpreter (MAI) configuration and DRAM address interleaving dimensions.

A. Memory Access Granularity

As discussed in Section III, with *rns_concat* based schemes, consecutive addresses do not map onto contiguous memory locations. This causes spatial locality in caches to suffer, prompting us to salvage sequential locality by increasing the memory access granularity to that of a cache line. Under this paradigm, each memory access now refers to a 64-byte chunk of memory rather than a single byte. Clearly, this requires support from the software stack, and we propose an ISA extension, which we name as the SELECT instruction, to help.

The SELECT instruction takes as arguments a base address (represented as *rns_concat*), and a separate offset represented in binary to perform a word lookup within a cache

line. This hybrid binary-*rns_concat* representation renders the best of binary on one hand, i.e., sequential locality as intended by the programmer, and that of *rns_concat* on the other hand, i.e., no runtime conversion overhead. A detailed example of how such a compiler transformation may be effected is presented in Section VIII for a word size of 8 bytes, although other word sizes can also be supported.

Representing a portion of the address in binary may seem counter-productive to the reliability of the system, but note that this offset is static (set at compile time) and therefore does not warrant computational error correction, meaning that the ECC protection that comes naturally to instructions (Section II-C) is sufficient to protect this offset as well.

Introducing such a representation comes with a set of limitations in software. First, if each cache line access is to be accompanied by a SELECT instruction, a static code bloat of about 15% occurs. In practice, we expect this bloat to decrease significantly due to spatial locality; once a cache line is loaded, multiple SELECTs can be performed without re-issuing the cache line access. Static code bloat can also be reduced by designing compiler optimizations that further leverage spatial/temporal locality in instruction layout and scheduling. Second, an increased memory access granularity renders arbitrary pointer arithmetic and branch targets tricky to disambiguate at compile time, unless they become aligned to cache-line boundaries. Finally, for general purpose system integration, such a hybrid representation may be required to be extended into the software runtime to avoid an explosion of pages. Alternately, TLB and paging performance may also be improved by employing super pages [72], [100], especially in emerging storage class memories [10] (with segmentation support for security).

The SELECT instruction requires tight integration with the software stack, the detailed implementation of which is beyond the scope of this paper.

B. Memory Address Interpreter (MAI)

Figure 2 presents a simplified view of the memory system. With typical memory systems comprising of L1, L2, L3 caches and a DRAM, the Memory Address Interpreter does not necessarily have to be a singleton unit at the highest level of cache. With a non-inclusive cache hierarchy, the following exhaustive set of legal Memory Address Interpreter configurations are explored:

1) **Ideal**

IBIN This is the ideal configuration where conversion to binary is without any overhead, or equivalently, a non-error-correcting core that uses a conventional weighted binary representation.

2) **Naive**

NL1 This is a naive conversion approach where every memory access is converted to binary before accessing the L1 cache, thereby incurring a conversion overhead of 8 cycles and its associated energy consumption (adders and lookup tables).

NRNS This is a naive conversion approach where the tuple of residues in the MAR is simply concatenated (*rns_concat*).

NMEM This is a naive conversion approach where *rns_concat* is used through the cache hierarchy and a conversion to binary is effected at the memory controller, thereby incurring an overhead of 24 core cycles (this is because the clock domain of the memory controller is nominally about three times slower than that of the core).

3) **Compiler.** These approaches require compiler support:

CRNS This uses compiler support to realize a memory access granularity of 64 bytes (cache line) and simply concatenates the tuple of address residues before accessing L1. In other words, this is **NRNS** when the SELECT instruction is used.

CX **X**$\in\{$**L2**, **L3**, **MEM**$\}$. These are similar to **CRNS**, except that a conversion to binary is effected before accessing the L2/L3/main memory respectively.

4) **Rns_sub**

SUB This employs the single cycle overhead conversion of *rns_sub* before the L1 cache is accessed.

SUBM This is similar to **SUB** except that a binary conversion is effected before a main memory access.

5) **CLB**

CLBX **X**$\in\{$**L1_N**, **MEM_N**$\}$. This is similar to **NL1** or **NMEM**, except that an **N**-entry CLB is used in an attempt to hide the performance overhead of the conversion to binary, although at a potentially higher energy cost. Upon a CLB hit, a single cycle conversion is rendered, however, a CLB miss renders an access time equal to that of a binary conversion as usual. Furthermore, **CCLBMEM_N** is similar to **CMEM** but with an **N**-entry CLB at the memory controller.

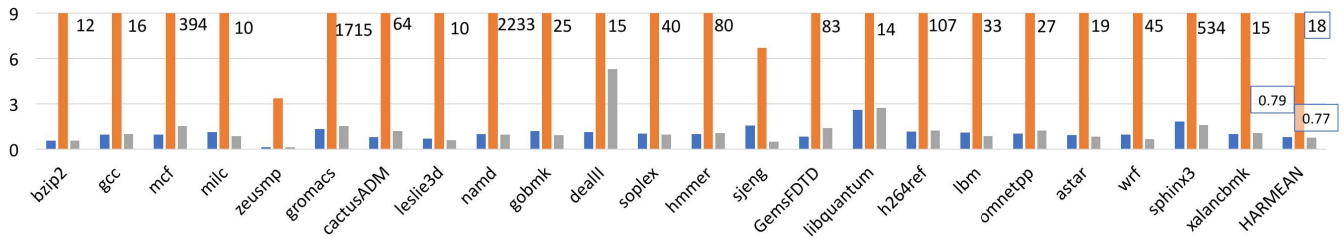
CLBY **Y** is of the form **S_N**. This is similar to **SUBM**, except that the conversion to binary at the memory controller is augmented with an **N**-entry CLB.

Space of valid configurations. Those listed above are deemed representative of an exhaustive brute force sweep. For example, it doesn't make sense to perform a conversion to RNS once a binary conversion or an *rns_sub* has already been effected. Also, mixing *rns_sub* at L1 and conversion to binary at L2/L3 imposes constraints on possible cache configurations[1], hence, we exclude them from our evaluation (although the careful reader may reason about these tradeoffs from the analysis and evaluation presented in this paper).

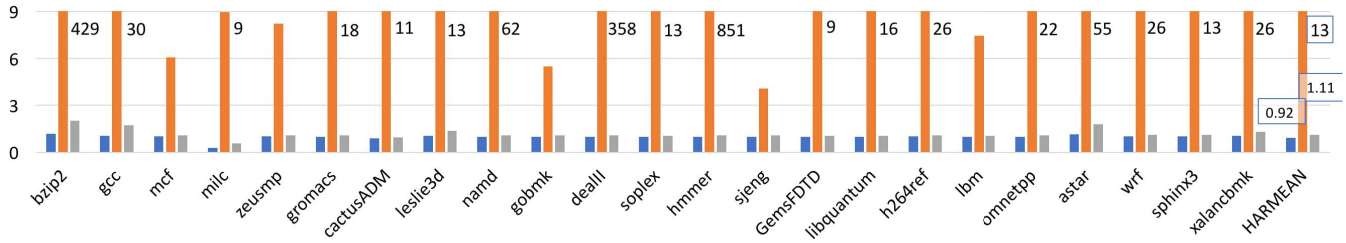
Cache coherence. Cache coherence state is typically maintained at the granularity of a cache line. With the exception of **NRNS**, **NMEM** and **CLBMEM_N**, all the other configurations proposed preserve locality within a cache line at the very least, when compared to **IBIN**. To elaborate, **Compiler** approaches are specifically designed to preserve sequential locality within a cache line. **NL1** and **CLB** based approaches that effectively convert an address to binary prior to L1 trivially preserve locality within a cache line. **Rns_sub** based approaches preserve locality for m_4 consecutive bytes in general (Section III-B); since all the moduli (and m_4, in particular) are greater than 63, therefore, **Rns_sub** preserves cache line locality as well. We conclude that all efficient and well performing configurations proposed are also amenable to traditional, unmodified cache coherence protocols.

Summary. Intuitively, we first observe that the **Compiler** approaches would benefit from the best of locality-preserving properties of *binary* and the no-overhead nature of *rns_concat*, however, at the cost of requiring changes to and tight integration with the software stack. Next, **Rns_sub** approaches would benefit from both its locality-preserving, as well as memory level parallelism inducing properties, at low cost. Finally, **CLB** based approaches would mimic

[1]For example, given the cache configuration in our evaluation, the addresses X (0x4fab84d8) and Y (0x4fab84fc) when subject to *rns_sub* at L1 and conversion to binary at L2, map onto the same L1 index (36) but different L2 indices (165, 166). This makes enforcing a generic non-inclusive property unnecessarily complex.



(a) L1 MPKI normalized to binary



(b) L3 MPKI normalized to binary

Figure 4: Misses Per Kilo Instructions (MPKI – lower is better) of representative RNS based schemes, normalized to a binary scheme. **SUB** successfully retains spatial locality present in the lower order bits of **IBIN**. **CRNS** salvages sequential locality lost by **NRNS**, with the help of compiler support.

IBIN, however, at a significant energy overhead. We will demonstrate in detail in Section V that the relative performance is **Ideal** > **Compiler** > **CLB** > **Rns_sub** >> **Naive**, with **Rns_sub** rendering the most energy efficient architecture along with the added advantage of requiring no support from the software stack.

C. DRAM Address Interleaving

The MAI of Section IV-B directly impacts DRAM address interleaving. While an industry standard memory controller policy is typically undisclosed, we span the following interleaving policies gathered from published research:

- **Row**: The LSB bits of the memory address decide the Column index, followed by the Row index, Bank, Rank and the MSB bits decide the Channel (ChRaBaRoCo).
- **Channel**: This address interleaving is devised with the aim of boosting memory level parallelism when a *binary* coded memory address is presented (RoBaRaCoCh).
- **MinOp**: This address interleaving splits the column indices such that exactly 4 consecutive cache lines (assuming a *binary* coded memory address) may map to a single row, thereby striking a balance between row buffer hit rate and memory level parallelism [46].
- **XOR_***: These apply a permutation-based interleaving scheme (for reasons similar to the above) to each of the interleavings presented above [85], [118]. The essence of this is to set the bank index by XORing the bank bits with a selection of higher order bits.

V. Experimental Evaluation

A. Evaluation Methodology

We use pin [65] to generate a dynamic instruction trace that records the program counter of each instruction, as well as any load/store address. Our pintool instruments all 32 bit x86 gcc (-O2) optimized binaries from the SPEC 2006 [20] benchmark suite, with test inputs. These traces drive a ramulator [54] based simulator, complete with an L1, L2, L3 non-inclusive cache hierarchy and DRAM main memory. The baseline configuration is presented in Table II.

Table II: Baseline Configuration

Parameter	Dimensions
Core	Inorder / OoO
OoO Fetch/Retire width/ROB size	4/4/128
L1 size/associativity	32kB/8-way
L2 size/associativity	256kB/8-way
L3 size/associativity	2MB/8-way
Load to use latency L1/L2/L3	4/4+12/4+12+31 cycles
MSHR per cache	16
Caching policy	Non-inclusive/LRU
Core-Memory frequency ratio	3:1
DRAM JEDEC Standard	DDR4 (1channel) / LPDDR4 (2ch)
DRAM address interleaving	Section IV-C
DRAM policy	FRFCFS_prioritizeHit Open page
Memory Address Interpreter	Section IV-B
CLB size	128/1024 entries
CLB policy	Fully associative/LRU
CLB hit / miss latency	1 cycle/binary conversion

We enhance the simulator to support the design space presented in Section IV, and use McPat [59]/CACTI [60] to model energy overheads due to the Memory Address Interpreter. Recall from Section II-C that memory addressing logic such as the MAI is assumed to be error-free and can therefore be modeled with conventional MOSFET-based circuits. For the purposes of the power and area model for the MAI, we assume a 32nm/300K/0.9V/2GHz configuration.

B. Cache

First, we demonstrate the intuition formed in the theoretical analysis of Section III regarding the impact RNS addressing schemes have on locality. Figure 4 shows the cache misses per kilo instructions (MPKI) of RNS schemes, when normalized to binary. *Binary*, captured by **IBIN**, is the normalizing baseline; *rns_concat* is captured by **NRNS**, with the effect of introducing the SELECT instruction captured via **CRNS**, and **SUB** captures behavior of *rns_sub*. The other MAI configurations from Section IV-B do not introduce any new addressing schemes.

As expected from Section III, **NRNS** suffers a significant (18×/13× for L1/L3) loss in spatial locality, whereas **CRNS** helps salvage this. **SUB** was designed to retain the spatial

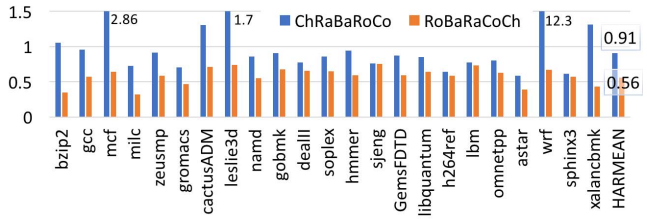
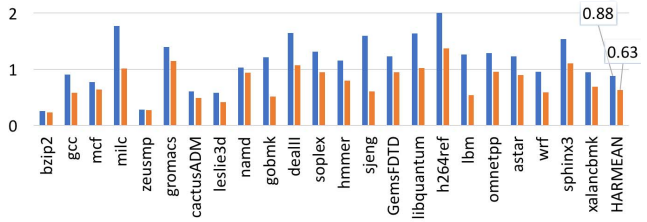
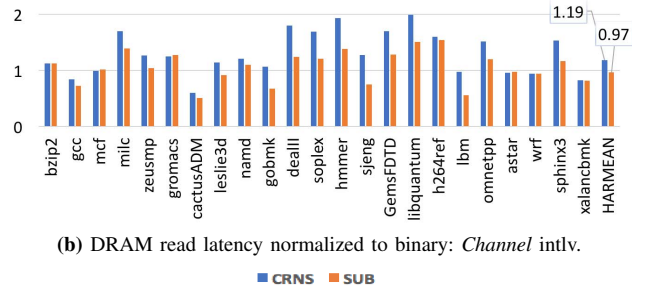


Figure 5: DRAM row buffer hit rate (higher is better) of **SUB**, normalized to **IBIN**, under two example address interleavings. Row locality is completely lost in **CRNS** irrespective of interleaving, and is therefore omitted from this plot. Row locality for **SUB** is relatively less impacted, but prefers the first interleaving, by design.



(a) DRAM read latency normalized to binary: *Row* interleaving.



(b) DRAM read latency normalized to binary: *Channel* intlv.

Figure 6: DRAM read latency (lower is better) of representative RNS based schemes, normalized to a binary scheme, under two example address interleavings. RNS based schemes naturally extract more memory level parallelism. Together with row locality characteristics (Figure 5), read latency of RNS based schemes are about on-par with or better than binary based schemes.

locality that is present in the lower order bits of **IBIN**, and successfully, is more or less on par with it. The interplay of addressing and temporal locality is rendered responsible for cases where RNS schemes show superior cache performance to binary. Results from cache configuration sweeps w.r.t. size/associativity/replacement policy are as expected. Therefore, we omit them as it adds no new insight to the architecture community.

C. DRAM

There are two aspects to DRAM performance: row buffer hit rate and memory level parallelism exploited.

By design, we expect **SUB** to show somewhat similar row buffer hit rate to **IBIN** for an interleaving policy that respects the locality preserving design principle of **SUB**, such as *row* interleaving. Figure 5 shows its hit rate, normalized to that of binary, for two example interleaving policies. **CRNS**, however, preserves spatial locality at a smaller granularity, thereby exhibiting very low row buffer hit rate. For brevity, we do not present detailed results for **NRNS** because of its incredibly poor cache performance, and omit depiction of **CRNS** from Figure 5 because of its near zero hit rate. The interleaving policy has a significant impact on row buffer locality.

Table III: Most favored DRAM interleaving policy for an MAI configuration. In general, RNS based configurations (such as **NRNS**) benefit from increased memory level parallelism especially when linear address interleaving such as *row* interleaving are used and binary based configurations (such as **IBIN**) benefit from permuted and *XOR* interleavings. Favorable interleavings choice is also affected by row buffer locality and memory pressure differences (due to interplay between MAI configuration, cache performance and processor configuration). If several equally performant choices are available, an arbitrary selection is made.

	MAI Configuration	Inorder	OoO
Ideal	IBIN	XOR_MinOp	XOR_MinOp
Naive	NL1	XOR_Channel	
	NRNS	Row	Row
	NMEM		
Rns_sub	SUB	XOR_MinOp	XOR_MinOp
	SUBM	Row	Row
Compiler	CRNS	XOR_MinOp	XOR_MinOp
	CL2	Row	Row
	CL3	XOR_MinOp	XOR_MinOp
	CMEM		
Compiler / CLB	CCLBMEM_128		
	CCLBMEM_1024		
CLB	CLBL1_128		
	CLBL1_1024	XOR_Channel	
	CLBMEM_128	Row	Row
	CLBMEM_1024		
Rns_sub / CLB	CLBS_128	XOR_MinOp	XOR_MinOp
	CLBS_1024		

On the other hand, because RNS addressing *naturally* permutes an address, a higher degree of memory level parallelism is expected, especially for linear interleaving policies such as *row* interleaving. Figure 6 demonstrates this, as we observe that the average read latency is significantly improved inspite of an inferior row buffer hit rate for RNS based schemes. This is one of the reasons why several interleaving policies (Section IV-C) are deployed for binary based addressing systems, i.e., permuting the address bits reduces bank conflicts and therefore boosts performance (other reasons not related to performance improvement for such permutation are to avoid row-hammer [2] and security issues).

The results presented in this section are with an inorder processor, but similar trends are seen for OoO as well. Also, DRAM scheduling and paging policies have an insignificant impact on performance when compared to the impact due to address interleaving. Therefore we omit their results for brevity.

D. Overall Runtime

For completeness, we simulate the exhaustive set of 18 MAI configurations from Section IV-B across all 6 DRAM address interleavings from Section IV-C, and for presentation purposes summarize the performance for each configuration with its most favored DRAM interleaving policy (Table III) in Figure 7. In deriving the most favored interleaving, no distinction is made between workloads, i.e., the interleaving policy is varied only with MAI configuration and is workload independent.

We normalize their performances against a fixed baseline (**NL1** with *row* interleaving) to be able to compare across configurations and interleavings, and present the resulting speedups. We present only the harmonic mean across the benchmark suite.

[**Ideal**] For example, a conventional non-error-correcting binary core **IBIN** performs 2.84×/2.13× faster on average when compared to the baseline (for inorder/OoO processor respectively, with a DDR4 DRAM), and favors XOR_MinOp as its preferred address interleaving.

[**Naive**] None of the alternate address interleavings make a noticeable improvement to the performance of the baseline

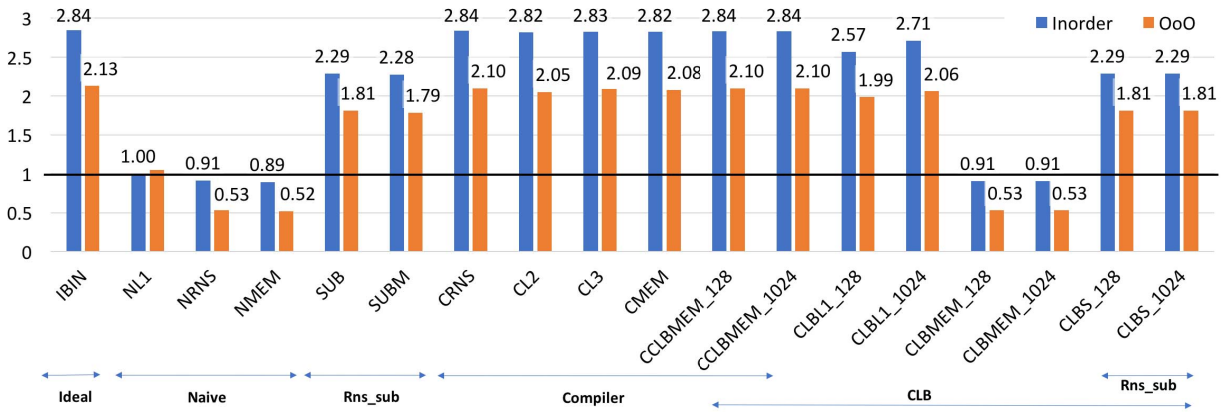
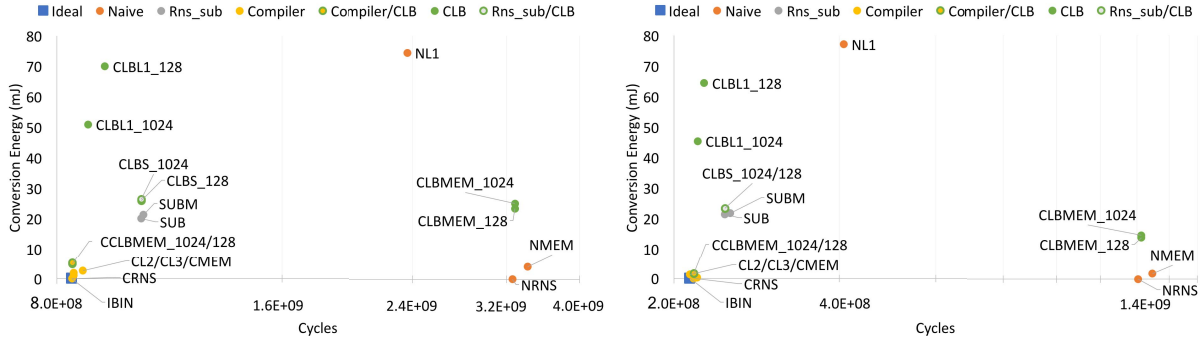


Figure 7: Speedup (higher is better) when compared to a naive approach of converting to binary upon each L1 access (**NL1** with *row* interleaving). Inorder cores are understandably more sensitive to conversion latency than OoO cores. For both inorder/OoO cores, the relative performance of each cluster is **Ideal** > **Compiler** > **CLB** > **Rns_sub** >> **Naive**.



(a) Inorder processor

(b) OoO processor

Figure 8: Overhead in conversion energy (mJ) vs runtime; lower is better for both axes (similar to a pareto chart). **SUB** is the most efficient microarchitecture configuration that requires no support from the software stack.

NL1. The 8 cycle conversion latency on every access to the memory hierarchy is left unhidden, allowing it to dominate the performance trends. **NRNS/NMEM** incur significant slowdowns because of their poor cache performance.

[**Rns_sub**] **SUB** shows significantly improved performance over the naive approaches. **SUBM** follows closely behind, due to slight decrease in exploited memory level parallelism, coupled with its conversion latency at the memory controller.

[**Compiler**] The approaches that leverage the SELECT instruction effectively approach **IBIN** by combining the best of binary (cache performance) and RNS (no conversion overhead).

[**CLB**] Placing a 128 entry CLB before L1 allows for performance superior to **SUB**, and a 1024 entry CLB seems sufficient to approach the performance of **IBIN**. However, placing a CLB at the memory controller is rendered useless unless either the SELECT instruction is used or *rns_sub* is used to prevent an explosion of cache misses.

From an Amdahl's law perspective, the amount of overhead and variation in memory access patterns introduced by various MAI configurations has more weight than just the DRAM interleaving policy, which is to be expected. If we were to choose an unfavorable DRAM interleaving instead, the performance on average for each of the non-naive configurations vary by less than 1%, although we omit detailed results for brevity.

LPDDR4 is a likely candidate to be used in conjunction with low power microarchitectures, but comes at a cost of increased row activation latency. Nevertheless, we find that the insights presented in this paper hold even when an LPDDR4 DRAM is used.

E. Energy Overhead

Figure 8 presents the performance of each of the MAI configurations, when put in perspective of how much *additional* energy they consume. For each MAI configuration, their most favored DDR4-DRAM interleaving is chosen (Table III) and the mean cycle count and conversion energy overhead (mJ) across the benchmark suite are presented.

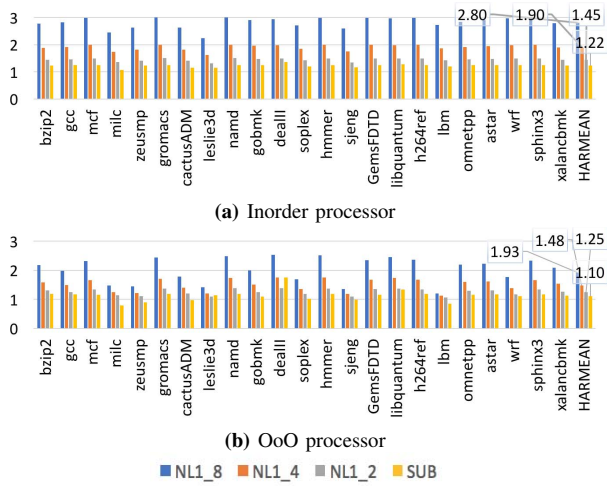
[**Ideal**] For example, at the bottom left is **IBIN**, taking the least number of cycles and without any conversion overhead.

[**Naive**] **NL1**, as has been mentioned throughout this paper, suffers from poor performance and high energy consumption. **NMEM/NRNS** have little or no increase in energy consumption due to conversion alone, but their performance suffers greatly.

[**Rns_sub**] **SUB/SUBM** are the most *efficient* microarchitectures that do not require support from the software stack for them to work.

[**Compiler**] While these approach **IBIN**, they are subject to software compatibility and integration issues as discussed in Section IV-A.

[**CLB**] While placing a CLB at the L1 is more performant than **SUB**, it comes at a higher conversion energy overhead. Placing a CLB at the memory controller must be accompanied by either an *rns_sub* addressed cache or a cache line addressed (SELECT instruction) cache.



(a) Inorder processor

(b) OoO processor

Figure 9: Slowdown of hypothetically faster binary convertors (4 cycles and 2 cycles as opposed to the best known candidate: 8 cycles) when compared to **IBIN**. **NL1** thats used everywhere in this document is annotated as **NL1_8** for clarity. Our most efficient microarchitecture technique that uses **SUB** is still faster than these hypothetically faster convertors. It is also more energy efficient than these, however, a quantitative comparison is not possible in the absence of concrete algorithms for faster binary conversion.

Deriving a CLB configuration is similar to that of a TLB- or cache-like structure. For example, while increasing the CLB size increases access power, it may reduce the number of CLB misses, which also translates into energy savings on CLB miss repairs (ex: **CLBL1_128** vs **CLBL1_1024**). For brevity, we limit a CLB configuration sweep to just these two for demonstrative purposes; the reader should be able to easily infer points of tradeoff for other CLB configurations.

MAI Area. Table IV presents the area requirements of few key configurations. We conclude that the energy trends discussed above apply to area as well.

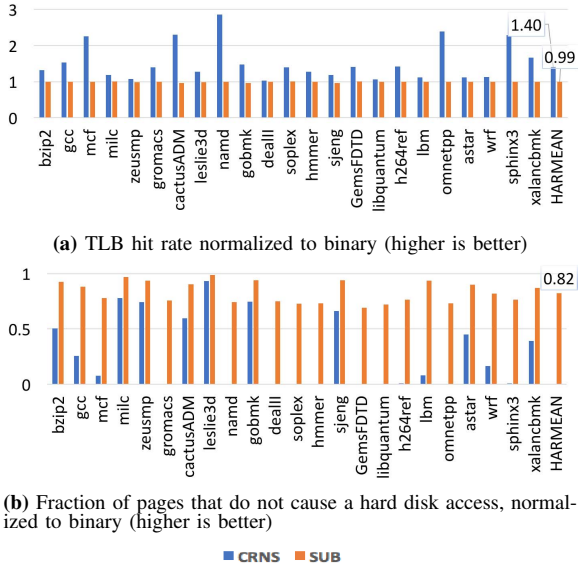
Table IV: Area requirement in mm^2.

NRNS	SUB	CLB_128	CLB_1024	NL1
0	0.075	0.091	0.104	0.160

F. Sensitivity to Conversion Latency

Throughout this paper, we have assumed an 8 cycle algorithm for converting an RNS tuple to a binary number (**NL1**), as it is the state of the art given that the bases must be amenable to RRNS error correction (Section II-B). Nevertheless, we also evaluate the relative performance of hypothetical conversion algorithms that take half as many or even a quarter of the cycles. For clarity, we differentiate these via subscripts (**NL1_8**, **NL1_4**, **NL1_2**). Figure 9 puts their performance in perspective of two representative MAI configurations: **IBIN** and **SUB**. Recall that **SUB** presents the most efficient microarchitecture that does not impose restrictions on software. Therefore, for this analysis, we choose **IBIN** as the baseline and also present **SUB** and **NL1_8** to put the performance of these hypothetical convertors into perspective. Relative performance of the remaining MAI configurations can be easily inferred from the previous result sections.

Not only are these hypothetical faster convertors less performant than the schemes presented in this paper, but would also be less energy efficient. While a quantitative comparison is not possible in the absence of concrete faster algorithms, we expect their energy overhead to be rather close to **NL1_8** (and certainly much more than **SUB**) from a functional standpoint of the process of converting to binary.



(a) TLB hit rate normalized to binary (higher is better)

(b) Fraction of pages that do not cause a hard disk access, normalized to binary (higher is better)

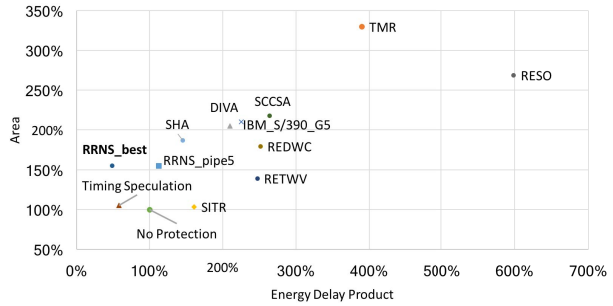
Figure 10: Virtual memory performance of **CRNS** and **SUB** when compared to **IBIN**. **CRNS** has superior TLB hit rate owing to its higher memory access granularity, however, an explosion is seen in the number of physical pages it touches, owing to its limited spatial locality granularity. As expected, both TLB and page pressure of **SUB** approach that of **IBIN**.

G. Virtual Memory

Certain RRNS based architectures may find it necessary to integrate with a virtual memory (VM) subsystem. There are two aspects to VM performance: TLB hit rate and page pressure. Under an idempotent virtual-to-physical transformation, we use the reuse distance [7], [12] mechanism to estimate the TLB hit rate in a 512kB structure and to estimate page pressure using an 8GB DRAM, assuming a page size of 4kB (as is common on Linux systems). We quantify page pressure by measuring the number of pages that have already been brought into main memory as well as the number of pages that need to access secondary memory (such as a non-volatile disk), as they may either have not yet been allocated a physical page frame or had been evicted to make way for newer pages.

Figure 10 highlights these for representative RNS schemes, when compared to binary. Given its 64 byte access granularity, **CRNS** exhibits high TLB performance thanks to sequential locality in programs. However, this granularity is rather small at the page level, causing an explosion in the number of pages it accesses. As described in Section IV-A, **CRNS** would have to be extended into the runtime system (or a different page granularity should be used) in order to attain low page pressure.

SUB, on the other hand, performs very similar to **IBIN** on both counts. The reason for its relatively higher page pressure when compared to binary is that 4kB pages have an offset of 12 bits, whereas the residues in this paper are (or specifically the fourth residue) 8 bits wide, thereby leading to a gap in locality between *rns_sub* and *binary*. Choosing a wider m_4 would result in an interesting tradeoff as it may change the power-performance-reliability characteristics of the RRNS core, but from a memory systems point of view, it would enhance the spatial locality properties of *rns_sub* which directly translates to improved cache performance, DRAM row buffer hit rate and virtual memory performance.



RESO [78], REDWC [45], RETWV [41], SCCSA [108], SHA [79], DIVA [1], SITR [70], Timing Speculation [26], [37]

Figure 11: 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.

VI. Related Work

Computational error correction. Figure 11, Section I-A summarize various techniques in comparison with RRNS. We refer the interested reader to Srikanth et al. [91] for a more detailed survey on some of these non-residue techniques; RRNS is generally considered superior in terms of capability and efficiency for computational error resilience. State of the art adoption and research in the industry also claim the superiority of residue based resilience [16], [61].

Approaches that employ timing speculation [26], [37] may seem superior to RRNS at first glance. However, the error model that can be supported by an RRNS error correcting architecture is orthogonal to theirs, if not broader. For example, razor [26] uses conventional transistors, therefore lowering V_{dd} lowers MOSFET switching speed significantly, resulting in a large 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 [37] uses a delayed commit approach (with rollback support) to handle violations in timing margins. However, with emerging devices (Section I), V_{dd} can be lowered to few tens of millivolts with a relatively lower frequency loss, meaning that operating at the resultant thermal noise floor leads to *stochastic, intermittent* bit flips, which cannot always be captured as circuit timing errors. Unlike such approaches, RRNS error correcting architectures 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 [1] 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 POWER6/7/8 and z10/z196 processors [8], [16], [44], [61] and various Intel Corporation [90] and Sun Microsystems based mainframes [43]. On the other hand, RRNS is able to tolerate errors in its redundant as well as non-redundant computations.

Finally, a vast majority of these related work are at the circuit level and can be augmented into RRNS-based architectures for potentially enhanced reliability characteristics.

Prime number based indexing. Kharbutli et al. [53] propose using prime numbers for cache indexing to reduce conflict misses. However, such indexing introduces fragmentation that can only be amortized by higher cache capacities. They therefore recommend using their technique only for larger caches (such as L2/L3). Furthermore, their technique isn't applicable to DRAM addressing.

VII. Conclusion

New logic devices enable reduction of the supply voltage to few tens of millivolts, yet maintaining a switching speed superior to MOSFETs under low power operation. However, they are subject to intermittent, stochastic bit errors in logic due to proximity of operation to the kT noise floor. Computational error correction using the RRNS representation is a promising approach to using these devices. However, prior RRNS based architecture studies assume a simple, unrealistic memory hierarchy without considering issues with RNS-based memory addressing. This research found that naive approaches such as **NL1** and **NRNS** incur an overhead of $2\times$-$4\times$ on average. We propose new conversion strategies and architecture extensions to significantly improve on naive memory system performance. In our classification of the design space, the relative performance is **Ideal** > **Compiler** > **CLB** > **Rns_sub** >> **Naive**, with **Rns_sub** rendering the most energy efficient architecture along with the added advantage of requiring no support from the software stack.

The careful analysis of RNS-based memory access pattern behavior and the detailed cost benefit analysis of the resulting design space presented in this paper enables and extends RRNS-error-correcting core microarchitectures built from ultra-low energy logic devices. Using such devices has the potential to punch through the power wall to restart single core performance scaling via architectural advances abandoned at the end of Dennard scaling a decade ago.

VIII. Appendix: SELECT Instruction

Consider an implementation with a 64 byte cache line size, a *double* of size 8 bytes such that the following *struct* has a size of 16 bytes, and the following snippet of code, where $M < N$ are arbitrary natural numbers that are not necessarily compile-time constants.

```
typedef struct { double x; double y; } Foo;
Foo foos[N];
double sum = 0;
for (i = 0; i < M; ++i) {
  sum += foos[i].x;
  sum += foos[i].y;
}
```

A pseudo-assembly code of the loop body in a *binary* computer would be as follows:

```
LD X, (foos + i*16)     // Read foos[i].x
LD Y, (foos + i*16 + 1*8)  // Read foos[i].y
ADD S, S, X
ADD S, S, Y
```

When SELECT instruction is used, the code transforms to:

```
uint8_t nOffset = sizeof(CACHELINE)/sizeof(Foo); //
     64/16=4
// ++i is in RRNS ; ++n is in binary
for (rns_t i = 0; i < M / nOffset; ++i) {
  LD A64, RNS(foos + i*sizeof(CACHELINE))
  for (int_t n = 0; n < nOffset; ++n) {
    SELECT X, A64, sizeof(Foo)*n
    SELECT Y, A64, sizeof(Foo)*n + 1*8
    ADD S, S, X
    ADD S, S, Y
  }
}
```

ACKNOWLEDGMENT

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under Contract DE-NA0003525.

REFERENCES

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

[2] K. Bains, J. Halbert, C. Mozak, T. Schoenborn, and Z. Greenfield, "Row hammer refresh command," Aug. 25 2015, uS Patent 9,117,544.

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

[4] J.-C. Bajard, L.-S. Didier, and J.-M. Muller, "A new euclidean division algorithm for residue number systems," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 19, no. 2, pp. 167–178, 1998.

[5] 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.

[6] 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.

[7] B. T. Bennett and V. J. Kruskal, "Lru stack processing," *IBM Journal of Research and Development*, vol. 19, no. 4, pp. 353–357, 1975.

[8] M. J. Boersma and J. Haess, "Residue-based error detection for a processor execution unit that supports vector operations," Mar. 17 2015, uS Patent 8,984,039.

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

[10] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 449–464, 2008.

[11] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 3. ACM, 1991, pp. 276–286.

[12] C. CaBcaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proceedings of the 17th annual international conference on Supercomputing*. ACM, 2003, pp. 150–159.

[13] B. H. Calhoun, A. Wang, and A. Chandrakasan, "Modeling and sizing for minimum energy operation in subthreshold circuits," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 9, pp. 1778–1786, 2005.

[14] B. Cao, C.-H. Chang, and T. Srikanthan, "A residue-to-binary converter for a new five-moduli set," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 5, pp. 1041–1049, 2007.

[15] G. Cardarilli, M. Re, and R. Lojacono, "Rns-to-binary conversion for efficient vlsi implementation," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 45, no. 6, pp. 667–669, 1998.

[16] S. Carlough, A. Collura, S. Mueller, and M. Kroener, "The ibm zenterprise-196 decimal floating-point accelerator," in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*. IEEE, 2011, pp. 139–146.

[17] 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.

[18] 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.

[19] 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.

[20] S. CPU2006, "Standard performance evaluation corporation," 2006.

[21] 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 *Rebooting Computing (ICRC), IEEE International Conference on*. IEEE, 2016, pp. 1–8.

[22] 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, 1974.

[23] 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.

[24] 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.

[25] 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.

[26] 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, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 2003, pp. 7–18.

[27] 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.

[28] 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.

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

[30] D. Gamberger, "New approach to integer division in residue number systems," in *Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on*. IEEE, 1991, pp. 84–91.

[31] H. L. Garner, "The residue number system," *IRE Transactions on Electronic Computers*, no. 2, pp. 140–147, 1959.

[32] 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.

[33] 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.

[34] 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.

[35] J. González and A. González, "Limits of instruction level parallelism with data speculation," in *Proc. of the VECPAR Conf.* Citeseer, 1998, pp. 585–598.

[36] K. C. Gower, B. Hazelzet, M. W. Kellogg, and D. J. Perlman, "High reliability memory module with a fault tolerant address and command bus," Jun. 19 2007, uS Patent 7,234,099.

[37] 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, 2008. HPCA 2008. IEEE 14th International Symposium on*. IEEE, 2008, pp. 381–392.

[38] 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.

[39] A. A. Hiasat and H. Abdel-Aty-Zohdy, "Semi-custom vlsi design and implementation of a new efficient rns division algorithm," *The Computer Journal*, vol. 42, no. 3, pp. 232–240, 1999.

[40] M. A. Hitz and E. Kaltofen, "Integer division in residue number systems," *IEEE transactions on computers*, vol. 44, no. 8, pp. 983–989, 1995.

[41] 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.

[42] 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.

[43] S. Iacobovici, "End-to-end residue based protection of an execution pipeline," Jun. 30 2009, uS Patent 7,555,692.

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

[45] 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.

[46] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A dram page-mode scheduling policy for the many-core era," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 24–35.

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

[48] 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.

[49] R. Keyes, "Miniaturization of electronics and its limits," *IBM Journal of Research and Development*, vol. 32, no. 1, pp. 84–88, 1988.

[50] A. I. Khan, C. W. Yeung, C. Hu, and S. Salahuddin, "Ferroelectric negative capacitance mosfet: Capacitance tuning & antiferroelectric operation," in *Electron Devices Meeting (IEDM), 2011 IEEE International*. IEEE, 2011, pp. 11–3.

[51] A. I. Khan, K. Chatterjee, J. P. Duarte, Z. Lu, A. Sachid, S. Khandelwal, R. Ramesh, C. Hu, and S. Salahuddin, "Negative capacitance in short-channel finfets externally connected to an epitaxial ferroelectric capacitor," *IEEE Electron Device Letters*, vol. 37, no. 1, pp. 111–114, 2016.

[52] A. I. Khan and S. Salahuddin, "4 extending cmos with negative capacitance," *CMOS and Beyond: Logic Switches for Terascale Integrated Circuits*, pp. 56–76, 2015.

[53] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," in *Software, IEE Proceedings-*. IEEE, 2004, pp. 288–299.

[54] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[55] 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.

[56] 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.

[57] 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.

[58] H.-H. Lee, Y. Wu, and G. Tyson, "Quantifying instruction-level parallelism limits on an epic architecture," in *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on*. IEEE, 2000, pp. 21–27.

[59] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 469–480.

[60] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*. IEEE, 2011, pp. 694–701.

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

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

[63] 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.

[64] M. Lu and J.-S. Chiang, "A novel division algorithm for the residue number system," *IEEE Transactions on Computers*, vol. 41, no. 8, pp. 1026–1032, 1992.

[65] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

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

[67] 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.

[68] 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.

[69] A. McMenamin, "The end of dennard scaling," 2013.

[70] 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, 2007. SBAC-PAD 2007. 19th International Symposium on*. IEEE, 2007, pp. 45–53.

[71] P. V. A. Mohan, "Rns-to-binary converter for a new three-moduli set," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 9, pp. 775–779, 2007.

[72] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 89–104, 2002.

[73] 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.

[74] 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.

[75] 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.

[76] D. E. Nikonov and I. A. Young, "Overview of beyond-cmos devices and a uniform methodology for their benchmarking," *Proceedings of the IEEE*, vol. 101, no. 12, pp. 2498–2533, 2013.

[77] 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.

[78] 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.

[79] 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.

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

[81] 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.

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

[83] 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.

[84] 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.

[85] B. R. Rau, "Pseudo-randomly interleaved memory," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 3. ACM, 1991, pp. 74–83.

[86] M. Sachdev, "Fault-tolerant memory address decoder," Nov. 3 1998, uS Patent 5,831,986.

[87] S. Salahuddin and S. Datta, "Can the subthreshold swing in a classical fet be lowered below 60 mv/decade?" in *Electron Devices Meeting, 2008. IEDM 2008. IEEE International*. IEEE, 2008, pp. 1–4.

[88] 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.

[89] 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.

[90] Z. Sperber, O. Levy, M. Mishaeli, and R. Gabor, "Recoverable parity and residue error," Dec. 9 2014, uS Patent 8,909,988.

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

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] R. M. Swanson and J. D. Meindl, "Ion-implanted complementary mos transistors in low-voltage circuits," *IEEE Journal of Solid-State Circuits*, vol. 7, no. 2, pp. 146–153, 1972.

[97] 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.

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

[99] S. Talahmeh and P. Siy, "Arithmetic division in rns using galois field gf (p)," *Computers & Mathematics with Applications*, vol. 39, no. 5-6, pp. 227–238, 2000.

[100] M. Talluri and M. D. Hill, *Surpassing the TLB performance of superpages with less operating system support*. ACM, 1994, vol. 29, no. 11.

[101] 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.

[102] 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.

[103] ——, "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.

[104] T. N. Theis, "(keynote) in quest of a fast, low-voltage digital switch," *ECS Transactions, 45(6), 3-11*, 2012.

[105] T. N. Theis and P. M. Solomon, "In quest of the "next switch": prospects for greatly reduced power dissipation in a successor to the silicon field-effect transistor," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2005–2014, 2010.

[106] 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.

[107] 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.

[108] 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.

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

[110] J. Von Neumann, A. W. Burks *et al.*, "Theory of self-reproducing automata," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 3–14, 1966.

[111] 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.

[112] 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.

[113] 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.

[114] 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.

[115] 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.

[116] 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.

[117] 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.

[118] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. ACM, 2000, pp. 32–41.