J. Parallel Distrib. Comput. 97 (2016) 78–95

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

New distributed algorithms for fast sign detection in residue number systems (RNS)

Dhananjay S. Phatak *, Steven D. Houston

CSEE Department, UMBC, 1000 Hilltop Circle, Baltimore, MD 21250, USA

HIGHLIGHTS

- Most operations hitherto deemed hard to realize in RNS share the same bottleneck.
- Finding the LSB of an unknown in the Chinese Remainder Theorem is that bottleneck.
- Show 2 new fast methods to solve the bottleneck, one meets analytic speed bound.
- Moduli selection enables exhaustive pre-computation even at very long word lengths.

ARTICLE INFO

Article history:
Received 20 August 2015
Received in revised form
27 May 2016
Accepted 13 June 2016
Available online 6 July 2016

Keywords:
Residue number systems
RNS
Reconstruction coefficient
Partial reconstruction
Reduced precision
Fast sign detection

ABSTRACT

We identify a canonical parameter in the Chinese Remainder Theorem (CRT) and call it the "Reconstruction Coefficient", (denoted by "$\mathcal{R}_C$"); and introduce the notions of "Partial" and "Full" Reconstruction. If the $\mathcal{R}_C$ can be determined efficiently, then arithmetic operations that are (relatively) harder to realize in RNS; including Sign Detection, Base change/extension and Scaling or division by a constant can also be implemented efficiently. This paper therefore focuses on and presents two distinct methods to efficiently evaluate the $\mathcal{R}_C$ at long wordlengths. A straightforward application of these methods leads to ultra-fast sign-detection.

An independent contribution of this paper is to illustrate non-trivial trade-offs between run-time computation vs. pre-computation and look-up. We show a simple method to select the moduli which leads to both the (i) number of RNS channels n; as well as (ii) the largest channel modulus m_n satisfying $\{O(n)\ ,\ O(m_n)\} \lessapprox N \equiv$ the full-precision bit-length. The net result is that for many canonical operations; exhaustive look-up covering all possible input values is feasible even at long cryptographic bit-lengths N. Under fairly general and realistic assumptions about the capabilities of current hardware, the memory needed for exhaustive look-up tables is shown to be bounded by a low degree polynomial of n. Moreover, both methods to compute $\mathcal{R}_C$ can achieve a delay of $O(\lg\ n)$ in a RN system with n channels. To the best of our knowledge, no other method published to date has shown a path to achieve that lower bound on the execution delay. Further, small values of channel moduli make it ideal to implement each individual RNS channel on a simple core in a many-core processor or as a distributed node, and our algorithms require a limited number of inter-channel communications, averaging $O(n)$. Results from a multi-core GPU implementation corroborate the theory.

1. Notation and definitions

A residue number system (RNS) uses a set $\mathbb{M}$ of pairwise co-prime positive integers called the moduli-set:

$$\mathbb{M} = \{m_1, m_2, \ldots, m_r, \ldots, m_n\} \tag{1}$$

where each *component-modulus* $m_r > 1\ \forall r \in [1, n]$ and $\gcd(m_i, m_j) = 1$ for $i \neq j$. For convenience, we assume $m_i < m_j$ if $i < j$.

The number of residue *channels* n is the cardinality of the moduli-set:

$n \equiv |\mathbb{M}|$ = number of moduli.

The *total modulus* M is the product of all moduli in the moduli-set:

$\mathrm{M} = m_1 \times m_2 \times \cdots \times m_n$.

Every integer $Z \in [0, \mathrm{M} - 1]$ can be uniquely represented by a touple/vector of its residues (remainders) w.r.t (with respect to)

* Corresponding author.
E-mail address: phatak@umbc.edu (D.S. Phatak).

http://dx.doi.org/10.1016/j.jpdc.2016.06.005
0743-7315/

each component modulus:

$$Z \longleftrightarrow \overline{\mathbb{Z}} = [z_1, z_2, \ldots, z_n], \quad \text{where} \tag{2}$$

$$z_r = (Z \bmod m_r), \quad r = 1, \ldots, n. \tag{3}$$

The RNS *full-precision* is defined as N base-b digits, where $N = \lceil \log_b \mathrm{M} \rceil$ and b is the radix (or base) of the original representation of the integer Z.

Conversion from residues back to an integer is done using the *Chinese Remainder Theorem* (CRT) as follows:

$$Z = (Z_T \bmod \mathrm{M}) \quad \text{where} \tag{4}$$

$$Z_T = \left(\sum_{r=1}^{n} M_r \cdot \rho_r \right) \tag{5}$$

$$\rho_r = (z_r \cdot h_r) \bmod m_r \tag{6}$$

$$\text{with outer-weights } M_r : M_r = \frac{\mathrm{M}}{m_r} \tag{7}$$

$$\text{and inner-weights } h_r : h_r = M_r^{-1} \bmod m_r. \tag{8}$$

Note that the weights h_r and M_r are constants for a given $\mathbb{M}$.

The *reconstruction remainders* are defined as the per-channel ρ_r values defined by Eq. (6) and are therefore bounded:

$$0 \le \rho_r < m_r \quad \Rightarrow \quad 0 \le \frac{\rho_r}{m_r} < 1. \tag{9}$$

Eq. (4) can then be rewritten as

$$Z = Z_T - \mathcal{R}_{Cz} \cdot \mathrm{M} \quad \text{where} \tag{10}$$

$$\mathcal{R}_{Cz} = \left\lfloor \frac{Z_T}{\mathrm{M}} \right\rfloor = \text{the quotient of integer division of } Z_T \text{ by M}. \tag{11}$$

We define the *reconstruction coefficient* for integer Z to be the integer $\mathcal{R}_{Cz}$. For simplicity we also use only $\mathcal{R}_C$, dropping the variable-name-indicator when it is not needed. This canonical coefficient $\mathcal{R}_C$ satisfies magnitude bounds as explained below:

$$\frac{Z_T}{\mathrm{M}} = \sum_{r=1}^{n} \left(\frac{M_r \cdot \rho_r}{\mathrm{M}} \right) = \sum_{r=1}^{n} \left(\frac{\rho_r}{m_r} \right). \tag{12}$$

From relations (12) and (9), it follows that

$$0 \le \frac{Z_T}{\mathrm{M}} < n \quad \Rightarrow \quad 0 \le \mathcal{R}_{Cz} \le (n-1). \tag{13}$$

Full-reconstruction is defined as the generation of the entire unique digit-string for an integer, representing its numerical value in a non-redundant, weighted-positional format (such as the "sign-magnitude", "two's-complement" or the "mixed-radix format").

We define the *partial reconstruction* of an integer as evaluation of $\mathcal{R}_C$ for that integer without explicitly evaluating all digits in the full reconstruction.

2. Introduction

We would like to emphasize that we focus on RNS methods that can scale to bit-lengths commonly encountered in cryptography. The widely deployed RSA method uses one of the largest operand lengths, typically at least 1024 bits. We are therefore mainly interested in RNS methods that can scale to and work efficiently with operands that are 1024-bits or longer.

2.1. Advantages of RN systems and why they are ideal for cryptographic hardware

The main advantage of the RNS system is that in the Residue-Domain (RD), the operations in the set $\mathbb{S} = \left\{ \pm, \times, \stackrel{?}{=} \right\}$ can be performed in parallel in all n channels [30,19,23,12].

In other words, any of the operations in $\mathbb{S}$ on long word-length operands can be substituted by many smaller operations in channels (with operands that are no larger than the corresponding channel-modulus) that are completely independent of each other; and therefore can be performed in parallel. Note that equality of two numbers can be checked by comparing their residues which can be done in parallel in all channels. For this reason RN systems can be the ideal vehicles to implement cryptographic hardware with long word-lengths.

However, the extreme ease of implementing many of the basic arithmetic operations (those in the set $\mathbb{S}$) in the RNS is negated to a substantial extent by the fact that the following equally fundamental arithmetic operations are relatively harder to realize in the RNS [19,23,12]:

1. Full-reconstruction [19,23].
2. Sign detection or equivalently, magnitude comparison, or under/over-flow detection.
3. Base extension or a change of base.
4. Scaling or division by a constant, wherein, the divisor is known ahead of time (such as the modulus in the RSA or Diffie–Hellman algorithms).
5. Division by an arbitrary divisor whose value is dynamic, i.e., available only at run-time.

2.2. Full vs. partial-reconstruction

Reconstruction by straightforward application of the CRT (Eq. (4)) requires a remaindering with respect to M, which makes it slow (especially at long word-lengths used in cryptographic applications). To circumvent this difficulty, the CRT can be restated in an alternate form as an *exact integer equality* (see Eq. (10)). However, that relation contains an unknown: the Reconstruction Coefficient $\mathcal{R}_C$.

It turns out that a full-reconstruction is not necessary for any of the relatively harder operations that are listed above (except the full-reconstruction itself). If the value of $\mathcal{R}_C$ can be determined efficiently, then we can substitute the "exact-integer-equality" in lieu of the exact unique-digit-string for the operands, thereby avoiding a full reconstruction (or equivalently, an excursion out of the residue domain) which is costly since it requires a relatively large number of operations as well as a long delay. Therefore, in this paper, we show two highly distributed methods (the "Partial Reconstruction" algorithms) that allow a fast computation of $\mathcal{R}_C$ without necessarily performing a full reconstruction, and show how the value of $\mathcal{R}_C$ can be used to perform sign detection.

The rest of the paper is organized as follows: Section 3 presents a brief overview of prior and related work in RNS, clearly identifying their shortcomings. That naturally leads to an intuitive description of our new methods as well as their formal specification as algorithmic pseudo-code. Section 6 unveils an ultra-fast sign detection (or equivalently, a magnitude comparison) algorithm that can use any of the partial reconstruction algorithms. Performance analysis and comparison with existing methods can be found in Section 9.

3. Brief overview of prior and related work

Various aspects of RNS have been extensively studied for a while (approximately since [30,6]). For instance: for a sampling of works related to fast base change, see [27,17]; for a sampling of methods that use fractional intermediate computations, see [28,33,18].

Sign Detection or equivalently, magnitude comparison methods have also been extensively researched (ex, [30,6,33,21,8,20, 11,16,24,7,35,29,1,37]). A sampling of several other general improvements can be found in [15,9,25]. Efficient implementation

of full modular exponentiation within RNS has been studied as well (for a sample, see [2–5,13]).

Consistent with the title and scope of the paper (and to not exceed a reasonable length), this paper focuses only on Sign Detection (or equivalently, magnitude comparison methods). A comparison with other base-extension or change; modular reduction and other aspects are considered in the follow-on set of subsequent parts in separate papers.

Moreover, we restrict ourselves only to methods that can work with general/arbitrary moduli sets. Algorithms that use special moduli sets such as $\{2^{(m)}-1,\ 2^m,\ 2^{(m)}+1\}$ simply do not scale well to large cryptographic word lengths (for example, imagine a total modulus size of over 1000 bits with only three channels: in this case each channel would have operands of length $\approx \lceil \frac{1000}{3} \rceil =$ 334-bits, which is too long and slow). Hence, for the sake of fair and meaningful comparisons, only those methods that use non-specific general moduli are compared with the algorithms proposed in this paper (in Table 4).

3.1. The well-known *"extra-modulus" (integer domain only) method* to evaluate $\mathcal{R}_C$ by itself is not sufficient for sign-detection

Shenoy and Kumaresan proposed a fast and efficient base-extension method [27]. Therein, besides the residues for the original set $\mathbb{M}$, they require the residue of the target integer Z, with respect to one "extra/redundant" modulus m_e, i.e. $(Z \mod m_e)$ to be computed/available (we therefore refer to this method as the "Extra Modulus Method or (EMM)" in the rest of this paper).

Their method rearranges the statement of the CRT in the same form as in (10).

$$Z = Z_T - \alpha \cdot \mathcal{M}$$
$$\text{where } 0 \le \alpha \le n-1 \text{ (in our notation, } \alpha \triangleq \mathcal{R}_C) \tag{14}$$

where α is unknown. To evaluate α, they take the remainder (of both sides) of Eq. (14) with respect to the extra modulus m_e and use the known value of $(Z \mod m_e)$:

$$\alpha = [(1/\mathcal{M} \mod m_e) \times (Z_T \mod m_e - Z \mod m_e)] \mod m_e. \tag{15}$$

In order to be able to retrieve the value of α in this manner, the extra-modulus m_e must satisfy the following conditions:

$$m_e \ge n \quad \text{and} \tag{16}$$
$$\gcd(m_e, \mathcal{M}) = 1. \tag{17}$$

Once the value of $\mathcal{R}_C = \alpha$ is retrieved as per (15), then it can be plugged back into (14) to obtain an exact equality for Z. This works fine when it assumed that the value of $(Z \mod m_e)$ is available (note that this would be true if a freshly read-in binary number is converted to RNS and it is then required to extend or change the base. In that case, since the original value of Z was known, it is clear that the exact/correct value of $(Z \mod m_e)$ is also known; and therefore, the method works for base conversion).

However, when the RNS value Z_{RNS} is produced as a result of Addition or Subtraction, i.e.

$$Z_{\text{RNS}} = (X \pm Y) \mod \mathcal{M} \text{ then}$$

$$Z_{\text{RNS}} = \begin{cases} Z & \text{if there is no over/under flow, i.e. if} \quad 0 \le Z \le \mathcal{M} \\ Z - \mathcal{M} & \text{if there is an overflow,} \quad Z > \mathcal{M} \text{ and} \\ Z + \mathcal{M} & \text{if there is an underflow,} \quad Z < 0. \end{cases} \tag{18}$$

From relation (17), $(\mathcal{M} \mod m_e) \ne 0 \quad \Rightarrow$

$(Z_{\text{RNS}} \mod m_e) \ne (Z \mod m_e)$ whenever there is a wrap-around.

An overflow causes wrap-around from the right-hand side, whereas an underflow causes wrap-around from the opposite (i.e., left) side.

In other words, exact/accurate value of $(Z \mod m_e)$ (the second term in the numerator of the Right Hand Side (RHS) in Eq. (15)) is not always available after a subtract (or add) operation. Hence, EMM cannot be used to find the $\mathcal{R}_C$ of a number resulting from a subtraction. Consequently, the EMM is not sufficient to determine the sign (or compare two numbers in the residues format).

3.1.1. A small numerical example to demonstrate the insufficiency of the "extra modulus" method by itself for sign detection

The above fact can be demonstrated with the following simple numerical example:

Let the vector of moduli $\mathbb{M} = [2, 3, 5, 7] \Rightarrow \mathcal{M} = 210$.

The vector of outer-weights

(defined in Eq. (7)) $= [105, 70, 42, 30]$ and

the vector of inner-weights

(defined in Eq. (8)) $= [1, 1, 3, 4]$.

Finally, assume that the extra-modulus is $m_e = 11$.

Let two randomly selected integers be $X = 13$ and $Y = 44$.

RNS representation of $X \equiv \text{Vx} = [1, 1, 3, 6]$ and

$Y \equiv \text{Vy} = [0, 2, 4, 2]$.

Then, $Z = X - Y = 13 - 44 = -31$ so that

$Z_{\text{RNS}} = Z \mod \mathcal{M} = -31 + 210 = 179$.

Suppose that in the residue domain, vector Vz is calculated by a channel-wise modular subtractions of elements of Vx and Vy, so that $\text{Vz} = [1, 2, 4, 4]$.

The extra channel will evaluate

$$Z \mod m_e = (X \mod m_e - Y \mod m_e) \mod m_e = 2 \mod 11 = 2 \quad .$$

However, $(Z_{\text{RNS}} \mod m_e) = 179 \mod 11 = 3 \ne (Z \mod m_e)$

because of the wrap-around (underflow).

3.2. Drawbacks of known *Fractional Domain Method(s)* to evaluate $\mathcal{R}_C$

The CRT equation can be re-arranged in the following form:

$$\frac{Z_T}{\mathcal{M}} = \sum_{i=1}^{n} \frac{\rho_i}{m_i} = \mathcal{R}_C + \frac{Z}{\mathcal{M}} \quad \text{so that} \tag{19}$$

$$\mathcal{R}_C = \left\lfloor \sum_{i=1}^{n} \frac{\rho_i}{m_i} \right\rfloor = \text{Integer part of a sum of } n \text{ proper-fractions.} \tag{20}$$

The above idea of using the "fractional-representation" of CRT (i.e., the form shown in Eq. (20)), has been around for a while. However, to the best of our knowledge; all of the fractional domain methods and their derivatives/extensions that have appeared in the literature as of today suffer from one or both of the following drawbacks:

D-1. *Full precision fractional computations (with 1024 or larger number of fractional bits) are required*:

This can be done either by using indefinite-precision libraries for floating-point operations or using scaling and indefinite-precision libraries for integer arithmetic. Either way, such computations are obviously slower than operating on smaller word-lengths, and the full precision fractional bits require substantial storage.

For instance, Vu [34,33] proposed using a Fractional interpretation of the CRT in the mid 1980s. However, the method uses a very high precision of $\lceil \lg(n \cdot \mathcal{M}) \rceil$ bits (see equations 13 and (14) in

Ref. [33]). To the best of our knowledge to date, none of the subsequent follow-ons extending Vu's work have circumvented the need for such a large precision.

D-2. *Iterations on the order of the bit-length are needed*:

For example, the algorithm proposed in [18] and all of its derivatives as of today use a method to evaluate an approximate estimate $\widehat{\mathcal{R}_C}$ in a sequential, bit-by-bit (i.e., one bit-at-a-time) manner and then derive conditions under which the approximation is error-free. The iterative structure of this method makes it very slow at cryptographic word-lengths under consideration.

3.3. Additional methods that do not scale

Lu and Chiang [8,20] introduced a conceptually elegant method to use the "least significant bit" (`lsb`) to keep track of the sign. They start with the following observation:

If two integers X and Y have the same `lsb`, then $Z = (X \pm Y)$ is even.

Otherwise Z is odd.

Assume that the total modulus $\mathcal{M}$ is odd; $0 \leq \{X, Y\} \leq \mathcal{M} - 1$ and let Z_{res} be the residue domain representation of Z.

Then note that if $(X \pm Y)$ is out of the range $[0, \mathcal{M} - 1]$ then

$$Z_{\text{res}} = (X \pm Y \mp M) = Z \mp M \Rightarrow (Z_{\text{res}} \bmod 2) \neq (Z \bmod 2) \Rightarrow$$

if the `lsb` of Z is known, then

comparing it with the `lsb` of Z_{res} yields a method to detect whether a wrap-around (i.e. over or under flow) occurred; which, in-turn allows the determination of the correct sign of Z.

To see the close connection of this method with the extra modulus method, note that since $\mathcal{M}$ is odd, the condition in Eq. (17) is satisfied. Further, for any integer Z, `lsb` $(Z) = Z \bmod 2$. Hence, knowing `lsb` is tantamount to knowing the remainder with respect to the extra modulus $m_e = 2$.

It is therefore not surprising that these methods run into the same problem as all other EMMs: how to make sure that the value of `lsb` of actual (non-residue-domain true unique integer) value of Z is available in all cases?

In [8] the authors assume that they have an exhaustive look-up table for all possible inputs (see Table 1 on page 79 in Ref. [8]).

Such an exhaustive look-up table is not feasible for cryptographic word-lengths.

The second approach proposed by the authors in [20] is to resort to full precision computations of rational numbers (including fractions, see Eq. (9) and the implicated relations (which include the symbol "⇒") right below it, toward the end of Section 2, in column 1 on page 1029 in Ref. [20]). This is too slow at the long word-lengths under consideration as explained above.

Abtahi and Siy [1] also employ a method that assumes the value of `lsb` is known beforehand, although it is used in conjunction with core functions to help detect sign. Because of the reliance on the parity of Z, their method will fail if overflow/underflow occurred. Furthermore, it requires the pre-computation of specific parameters that would require full exhaustion over the entire RNS range $[0, 2^{N-1})$, which does not scale to large bit lengths.

Dimauro et al. [11] propose a novel technique of labeling the "diagonals" of the RNS space. The algorithm effectively maps each of the RNS numbers to a smaller set, and the label of the set can be used to compare the magnitude of two numbers. However, the method does not scale to large cryptographic bit-lengths, as either a modulo with respect to $\sum M_r$ must be computed, or a lookup table with $\sum M_r$ entries is required.

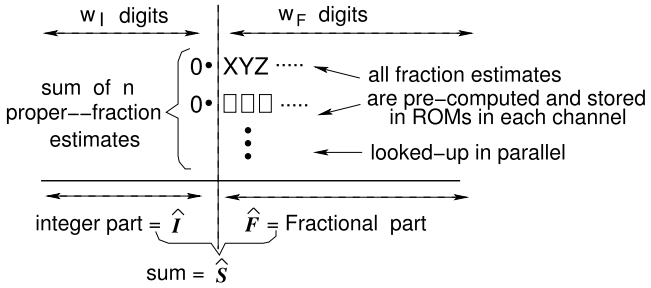


Fig. 1. Summation of drastically reduced precision fraction estimates (obtained by table look-ups) to estimate the reconstruction coefficient $\mathcal{R}_C$.

4. The novel reduced-precision partial reconstruction algorithm: RPPR

In this section, we develop a new reduced-precision partial reconstruction algorithm **RPPR_BASE**. It combines aspects of both Integer-domain as well as fractional-domain methods to circumvent all the difficulties illustrated in the previous section.

4.1. Narrowing the estimate of $\mathcal{R}_C$ with limited precision

Since $Z <$ M, it is clear from Eq. (10) that

$$\mathcal{R}_C = \left\lfloor \frac{Z_T}{\mathrm{M}} \right\rfloor = \left\lfloor \left(\sum_{r=1}^{n} f_r \right) \right\rfloor \quad \text{where} \tag{21}$$

$$f_r = \frac{\rho_r}{m_r} < 1. \tag{22}$$

Relation (21) states that $\mathcal{R}_C$ can be approximately estimated as the integer part of a sum of at most n proper fractions f_r, $r = 1, \ldots, n$, as illustrated in Fig. 1.

To speed up such an estimation of $\mathcal{R}_C$, we can leverage precomputations and look-ups of approximations to f_r. The look-up table for channel r (with modulus m_r) simply contains the values $\left[R\left(\frac{1}{m_r}\right), R\left(\frac{2}{m_r}\right), \ldots, R\left(\frac{m_r-1}{m_r}\right)\right]$, where R is a truncation to w_F digits. Note that if the reconstruction remainder $\rho_r = 0$, then the table entry is 0 which need not be explicitly stored.

Define

$$\widehat{f_r} = \text{truncation of } f_r \text{ to } w_F \text{ digits} \tag{23}$$

$$\widehat{S} = \left(\sum_{r=1}^{n} \widehat{f_r} \right) = \widehat{I} + \widehat{F}, \quad \text{where} \tag{24}$$

$$\widehat{I} = \lfloor \widehat{S} \rfloor \tag{25}$$

$$\widehat{F} = \widehat{S} - \widehat{I}. \tag{26}$$

The following result shows that the precomputed fractions f_r can be stored with drastically reduced precision, and $\mathcal{R}_C$ can still be confined to one out of two consecutive integers.

Theorem 1. *In order to narrow the estimate of $\mathcal{R}_C$ down to one out of two consecutive integers, $\widehat{I}$ and $\widehat{I} + 1$, it is sufficient to carry out the summation of the fractions in relation (24) in a fixed-point format with no more than $w_I = \lceil \log_b n \rceil$ digits for the integer part and $w_F = \lceil \log_b n \rceil$ digits for the fractional part.*

Proof. As seen in Fig. 1, our method simply looks up the estimates $\widehat{f_r}$ as specified by Eq. (23) and adds them together in a fixed-point format with a total precision of $w_I + w_F$ digits. Since the rounding mode is truncation, $\widehat{f_r}$ can only underestimate f_r, giving $0 \leq \widehat{f_r} \leq f_r < 1$. Since each summand satisfies $\widehat{f_r} < 1$, their sum $\widehat{S}$ satisfies $0 \leq \widehat{S} < n$. Thus, the integer part of $\widehat{S}$ satisfies $\lfloor \widehat{S} \rfloor \leq \widehat{S} < n$. This implies $\log_b(\lfloor \widehat{S} \rfloor) < \log_b n$, which proves the condition for w_I.

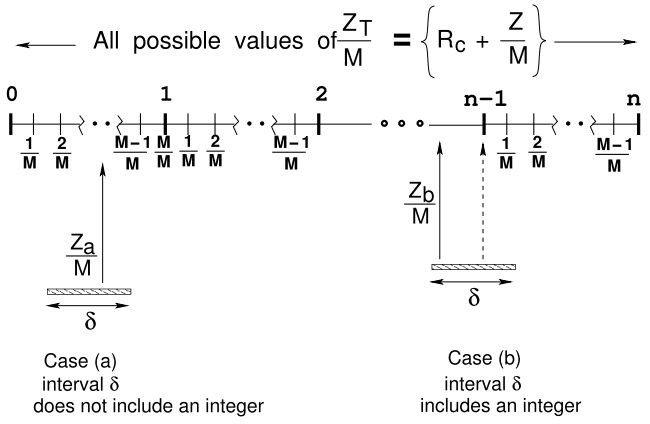


Fig. 2. Fundamental relation between δ and required precision.

Next, we demonstrate the sufficiency of precision of the fractional part. With truncation to w_F digits, the error ϵ_r between f_r and $\widehat{f_r}$ satisfies $0 \leq \epsilon_r = f_r - \widehat{f_r} < \frac{1}{b^{w_F}}$. This gives

$$\widehat{f_r} \leq f_r = \widehat{f_r} + \epsilon_r < \widehat{f_r} + \frac{1}{b^{w_F}}. \tag{27}$$

Summing relations (27) over all i from 1 to n we obtain

$$\widehat{S} \leq \sum f_r = \mathcal{R}_C + \frac{Z}{M} < \widehat{S} + \frac{n}{b^{w_F}}. \tag{28}$$

Then, the interval length δ of the error zone for $\sum f_r$ is

$$\delta = \left(\widehat{S} + \frac{n}{b^{w_F}}\right) - \widehat{S} = \frac{n}{b^{w_F}}. \tag{29}$$

In Fig. 2, it is clear that in order to narrow the estimate of the integer part of the sum to one out of two consecutive integers, the total length of the error zone (which is denoted by δ) must satisfy $\delta \leq 1$. This constraint is satisfied by imposing the condition

$$\frac{n}{b^{w_F}} \leq 1 \quad \text{or} \quad w_F \geq \lceil \log_b n \rceil . \tag{30}$$

Finally, taking the floor of each expression in (28) and using $\delta \leq 1$ we obtain

$$\lfloor \widehat{S} \rfloor \leq \mathcal{R}_C \leq \lfloor \widehat{S} + 1 \rfloor \tag{31}$$

giving

$$\widehat{I} \leq \mathcal{R}_C \leq \widehat{I} + 1. \quad \square \tag{32}$$

From Fig. 2, it can be seen that further discrimination among the two consecutive candidate values of $\mathcal{R}_C$ to select the correct one is possible or not-possible depending on whether or not the error zone straddles (i.e., includes) an integer:

(a) when the error zone δ does not straddle across an integer, then the value of $\mathcal{R}_C$ can be determined exactly as the smaller integer to the left of the error-zone (for example, the value 0 in Case (a) of the figure)
(b) if the error zone δ straddles across or includes an integer, then further disambiguation between the two candidate $\mathcal{R}_C$ values is necessary (for instance, $\{(n-2), (n-1)\}$ in Case (b) shown of the figure).

4.2. No-Ambiguity Zone

Even with the minimal precision stated in Theorem 1, it turns out that for a significant number of values of Z in the total range $[0, (\mathcal{M}-1)]$, we can determine the exact value of $\mathcal{R}_C$. The values of Z requiring no disambiguation of $\mathcal{R}_C$ tend to be clustered in a big region around the mid-point of the range, i.e., $\frac{\mathcal{M}-1}{2}$, while the values of Z requiring disambiguation of $\mathcal{R}_C$ between two consecutive integers tend to lie near the ends of the RNS range (toward 0 and M).

These properties are illustrated in Fig. 3. The x axes in the sub-figures show the range of a sample RNS system with $n = 6$ moduli in the set $\mathbb{M} = \{3, 5, 7, 11, 13, 17\}$. The Y axes in the figures show the frequency count of individual Z for which disambiguation of $\mathcal{R}_C$ is required. It is clear that both extremely small and large Z values (with respect to M) have a high probability of requiring disambiguation between two values of $\mathcal{R}_C$, while Zs near the midpoint of the RNS range have an extremely low probability. The sub-figures also show (and we will soon prove), that by slightly increasing the precision of the stored fractions (in this case from 3 to 6-bits), the vast majority of Z values require no disambiguation of $\mathcal{R}_C$.

If the value of the integer Z (which is being partially reconstructed) happens to lie in the regions filled with red colored bars, then Z may require extra computations to discriminate between two possible values for $\mathcal{R}_C$ (since the figures are frequency histograms, Z will require disambiguation or some nearby Z' will).

However, if Z is in the large white area (lacking histogram bars) in the center of the x-range in each of the sub-figures, then the error zone for Z's $\mathcal{R}_C$ calculation does not straddle an integer and therefore $\mathcal{R}_C$ can be determined without the need to further disambiguate.

Hence, we refer to the approximately "wide U" shaped (or concave bowl shaped) white area that includes the mid-point of the total range (i.e., $\frac{(\mathcal{M}-1)}{2}$) as the **No-Ambiguity Zone**, or the **NAZ**.

More precisely, for any given RNS (which in turn decides minimal precision and other parameter values) and a given instance of partial reconstruction algorithm, we define the **NAZ** as the largest interval $N_L \leq Z \leq N_H$, such that the algorithm can exactly determine the $\mathcal{R}_C$ for all Z in $N_L \leq Z \leq N_H$.

Increasing the precision leads to a smaller error zone, and a larger NAZ, which is experimentally demonstrated in Fig. 3: the word-length is increased from 3-bits (minimum required to narrow the estimate of $\mathcal{R}_C$ down to one out of two consecutive integers when $n = 6$) in the top sub-figure to 6 bits in the bottom sub-figure. It is clear that the frequency of ambiguation dramatically decreases and the size of the NAZ substantially increases with small increases in precision.

The following derivation is a generalization of Theorem 1 and gives a sufficient stored fraction precision for a given Z to lie in the NAZ.

Theorem 2. *If Z satisfies*

$$\frac{1}{\Phi} \leq \frac{Z}{M} \leq 1 - \frac{1}{\Phi} \tag{33}$$

where, Φ is an arbitrary precision parameter (or a constant) that satisfies the condition

$$0 < \frac{1}{\Phi} \leq \frac{1}{2} \tag{34}$$

then to determine the value of $\mathcal{R}_C$ exactly, it is sufficient to carry out the summation of the fractions in relation (24) in a fixed-point format with no more than $w_I = \lceil \log_b n \rceil$ digits for the integer part and $w_F \geq \lceil \log_b (n \cdot \Phi) \rceil$ digits for the fractional part.

Proof. The condition for w_I in Theorem 1. Eq. (28) gives us

$$\widehat{S} - \frac{Z}{M} \leq \mathcal{R}_C < \widehat{S} - \frac{Z}{M} + \frac{n}{b^{w_F}}. \tag{35}$$

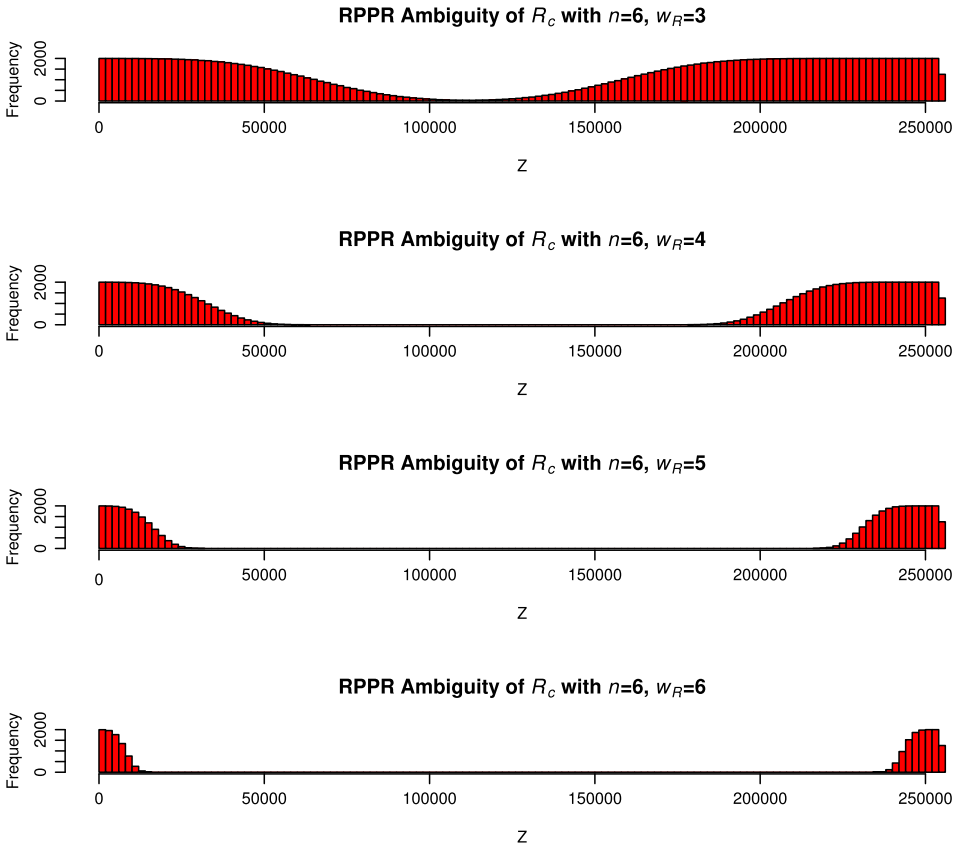


Fig. 3. Ambiguity of $\mathcal{R}_C$ in **RPPR**. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Adding the constraints of Theorem 2 gives

$$\widehat{S} - \left(1 - \frac{1}{\Phi}\right) \le \mathcal{R}_C < \widehat{S} - \frac{1}{\Phi} + \frac{n}{b^{w_F}}. \tag{36}$$

Since $\frac{1}{2} \le \left(1 - \frac{1}{\Phi}\right) < 1$, and $\frac{1}{\Phi} \ge \frac{n}{b^{w_F}}$, then

$$\widehat{S} - \kappa \le \mathcal{R}_C < \widehat{S}, \quad \text{where } \frac{1}{2} \le \kappa < 1. \tag{37}$$

Therefore, only one integer $\mathcal{R}_C$ can satisfy Eq. (37):

$$\mathcal{R}_C = \lfloor \widehat{S} \rfloor = \widehat{I}. \quad \square \tag{38}$$

By application of Theorem 2, by increasing the precision of the stored fractions by just 2 bytes beyond minimal precision ($w_F = \lceil \log_b n \rceil + 16$) yields an NAZ which occupies almost the entire range, i.e.;

$$\frac{\text{length}(\mathbf{NAZ})}{\mathcal{M}} \ge 1 - \frac{1}{32768}. \tag{39}$$

In other words, if Z is uniformly distributed in the range of the RNS, then less than one in 30,000 cases will we fail to return the exact value of $\mathcal{R}_C$ in a single shot. Consequently the expected or average delay of our **RPPR** algorithm is small.

4.3. Specification of the RPPR algorithm

Algorithm 1 presents a base **RPPR** method using Theorems 1 and 2 to reduce the possible values of $\mathcal{R}_C$ to at most two consecutive integers. Given an input Z and precomputations, it outputs the correct value of $\mathcal{R}_C$ or an approximation $\widehat{I}$ if $\mathcal{R}_C$ needs further disambiguation.

Algorithm 1: RPPR_base

Input : An integer Z in RNS form: $\overline{\mathbb{Z}} = [z_1, z_2, \ldots, z_n]$;
Precision parameter $\Phi > 2$ with $(\Phi \mod m_e) = 2$

Output: $\hat{R}$ and a binary flag Approx; if Approx $= 0$, then $\mathcal{R}_C = \hat{R}$; otherwise, if Approx $= 1$, then $\mathcal{R}_C$ equals either $\hat{R}$ or $\hat{R} + 1$

begin
- /* Per-channel memory stores precomputed approximations $\alpha_r = \lfloor 2^w \cdot \rho_r / m_r \rfloor$, where $w = \lceil \lg \Phi n \rceil$. */
- $c \leftarrow$ number of non-zero z_r with $r \le n$;
- $A_L \leftarrow \sum_{r=1}^{n} \alpha_r$ /* sum of pre-computed w-bit integers; scaled low estimate */
- $A_H \leftarrow A_L + c$ /* scaled high estimate */
- $a_L \leftarrow A_L / 2^w$ /* unscaled low estimate */
- $a_H \leftarrow A_H / 2^w$ /* unscaled high estimate */
- **if** $a_L == a_H$ **then**
  - **return** $\hat{R} = a_L$, Approx $= 0$;
- **else**
  - /* disambiguation is needed */
  - **return** $\hat{R} = a_L$, Approx $= 1$;

It turns out that if Z lies outside the NAZ, then Z can be iteratively multiplied by $(\Phi - 1)$ until the NAZ of the **RPPR_base** algorithm is reached. This iterative algorithm (denoted **RPPR**) is specified in Algorithm 2.

Lemmas 1 and 2 prove, respectively, that:

(1) Z and the multiplied result are congruent with respect to an extra modulus m_e, and

Algorithm 2: RPPR

Input : An integer Z in RNS form: $\overline{\mathbb{Z}} = [z_1, z_2, \ldots, z_n]$; Extra modulus m_e with $\gcd(M, m_e) = 1$; Precision parameter $\Phi > 2$ with $\Phi \mod m_e \equiv 2$.
Output: The correct value of $\mathcal{R}_C$.

```
begin
    Iter ← 1;
    R_C, Approx ← RPPR_base (Z̄, Φ);
    w_r ← z_r;
    R̂_C ← R_C;
    /* Iteratively multiply by (Φ − 1) and call
       RPPR_base until the NAZ is reached                    */
    while Approx == 1 do
        w_r ← ((Φ − 1)w_r) mod m_r ;
        Iter ← Iter + 1;
        R̂_C, Approx ← RPPR_base (W̄, Φ);
    if Iter == 1 then
        return R_C
    r_e ← R̂_C + M^{-1}(Σ ρ_{r,Z} M_r − Σ ρ_{r,W} M_r) mod m_e ;
    /* M^{-1} mod m_e is a constant known ahead of time
       or looked up                                          */
    /* Lookup ρ_{r,Z} M_r mod m_e and ρ_{r,W} M_r mod m_e,
       indexed by the values of z_r and w_r                  */
    if R_C mod m_e == r_e then
        return R_C
    else
        return R_C + 1
```

(2) since the exact value of $\mathcal{R}_C$ for the multiplied result can be determined, the exact value of $\mathcal{R}_C$ for Z can be determined. The only conditions are $\Phi \bmod m_e \equiv 2$ and $\gcd(M, m_e) = 1$, allowing us to choose an extra modulus $m_e = 2$, when the m_r of the moduli-set are odd and Φ is even.

Lemma 1. *If λ is the final value of* `Iter` *in Algorithm 2, with $W = (\Phi - 1)^{(\lambda-1)}\ Z \bmod M$, then, $(W \bmod m_e) = (Z \bmod m_e)$ when $\Phi \bmod m_e \equiv 2$.*

Proof. Let W_i denote the RNS number with residues w_r in the ith iteration of the while loop, where $W_i = \left[(\Phi - 1)^{i-1}\ Z \bmod M\right]$. Due to Theorem 2, when $\frac{M}{\Phi} \le W_i \le \frac{M(\Phi-1)}{\Phi}$, the value of $\mathcal{R}_C$ can be determined with a call to **RPPR_base** and the loop will exit.

Otherwise, let $W_{i+1} = (\Phi - 1)W_i$. Then $0 \le W_{i+1} < \frac{M(\Phi-1)}{\Phi}$ or $(\Phi - 2)M + \frac{M}{\Phi} \le W_{i+1} < (\Phi - 1)M \Rightarrow W_{i+1} \bmod M = (\Phi-1)W_i - cM$ where $c = 0$ or $c = (\Phi-2)$. Thus, $W_{i+1} \bmod m_e = W_i \bmod m_e = Z \bmod m_e$, when $(\Phi - 1) \bmod m_e = 1 \Rightarrow \Phi \bmod m_e = 2$. □

Lemma 2. *In Algorithm 2, an exact value of $\mathcal{R}_C$ can be determined for Z if $\Phi \bmod m_e = 2$ and $\gcd(M, m_e) = 1$.*

Proof. After the while loop, an exact value of $\mathcal{R}_C$ can be found for $W = (\Phi - 1)^{(\lambda-1)}\ Z \bmod M$ (since `Approx` $= 0$), and from Lemma 1, $W \bmod m_e = Z \bmod m_e$. We therefore have

$$\left(\left[\sum_r \rho_{r,Z} M_r\right] - M\mathcal{R}_{CZ}\right) \bmod m_e$$
$$= \left(\left[\sum_r \rho_{r,W} M_r\right] - M\mathcal{R}_{CW}\right) \bmod m_e$$
$$\Rightarrow \mathcal{R}_{CZ} \bmod m_e = \mathcal{R}_{CW} \bmod m_e + \left[\frac{1}{M} \bmod m_e\right]\left[\left(\sum(\rho_{r,Z} - \rho_{r,W})M_r\right) \bmod m_e\right]$$

which can be determined exactly if $\gcd(M, m_e) = 1$. Thus, since $\mathcal{R}_{C,Z}$ is known to be one of two consecutive integers, we can determine $\mathcal{R}_{CZ}$ exactly. □

It is further proven in Appendix A that the *maximum* number of iterations of **RPPR_base** in **RPPR** is bounded by $\lceil \log_{(\Phi-1)} M \rceil$ (Theorem 6). However, the *average* number of iterations for **RPPR** turns out to be a very small constant. For Z randomly chosen uniformly in $0 < Z < M$, if λ is the number of iterations required for each instance, then the average or expected value of λ satisfies

$$E[\lambda] < \frac{\Phi}{\Phi - 2} \quad \Longrightarrow \tag{40}$$
$$E[\lambda] = \Theta(1). \tag{41}$$

Please refer to Theorem 7 in Appendix A for further details.

4.4. Small numerical example illustrating the steps in *RPPR*

As a small example of **RPPR**, let $n = 4$ with the modulus set $m_1 = 3, m_2 = 5, m_3 = 7$, and $m_4 = 11$ and extra modulus $m_e = 2$. Then, the total modulus M $= 1155$, the outer-weights are $M_1 = 385, M_2 = 231, M_3 = 165$, and $M_4 = 105$, and the inner-weights are $h_1 = 1, h_2 = 1, h_3 = 2$, and $h_4 = 2$.

We will calculate the value of $\mathcal{R}_C$ for $Z = 10$, using the **RPPR** parameter $\Phi = 8$. We have residues $z_1 = 1, z_2 = 0, z_3 = 3$, and $z_4 = 10$, and a total of $\lceil \lg(n\Phi) \rceil = 5$ bits are required for each of the fractions (which is truncated scaled and stored in look-up table). Each channel looks up $\alpha_r = \lfloor 32 \cdot \rho_r / m_r \rfloor$, where $\rho_r = (h_r \cdot z_r) \bmod m_r$, giving $\alpha_1 = 10, \alpha_2 = 0, \alpha_3 = 27$, and $\alpha_4 = 26$. Summing the α_r, we get $A_L = 63$, and since there are three non-zero residues, $A_H = 66$. Shifting by 5 bits gives a low estimate for $\mathcal{R}_C$ of 1 and a high estimate of 2.

Since we must disambiguate, we multiply the z_r by $\Phi - 1$. The residues p_r of $\Phi - 1$ are $p_1 = 1, p_2 = 2, p_3 = 0$, and $p_4 = 7$. With $w_r = (\Phi - 1)z_r \bmod m_r$, we have $w_1 = 1, w_2 = 0, w_3 = 0$, and $w_4 = 4$. Again, each channel looks up α_r in its lookup-table memory, giving $\alpha_1 = 10, \alpha_2 = 0, \alpha_3 = 0$, and $\alpha_4 = 23$. Using the same procedure, we get $A_L = 33, A_H = 35$ and both a low and high estimate of 1 for $\mathcal{R}_C$. Since the upper and lower estimates converge to the same value, we have exactly determined the $\mathcal{R}_C$ of $W = (\Phi - 1)Z$. As a result, we exit the iterations-loop and back-calculate the exact value of $\mathcal{R}_C$ of Z.

Each channel now looks up it extra modulus information in the table; namely $\rho_{r,Z} M_r \bmod 2$ and $\rho_{w,Z} M_r \bmod 2$, respectively. We obtain $\sum(\rho_{r,Z} M_r - \rho_{w,Z} M_r) \bmod 2 = 1$. Since $M^{-1} \bmod 2 = 1$, we have $r_e = (1 + 1 \cdot 1) \bmod 2 = 0$. We now compare, r_e to the low estimate of $\mathcal{R}_C$ for Z. Since they are not equal, the exact value of $\mathcal{R}_C$ for Z is the high estimate of 2.

5. The mixed-radix partial reconstruction algorithm: MRPR

Even though the reduced-precision partial reconstruction algorithm **RPPR**, is very fast in the average case, it does require several iterations of the **RPPR_base** routine in the worst case. We now unveil a different novel algorithm that requires more precomputation and memory but limits the iterations to 1 in ALL cases.

Suppose that we pre-compute and store the value of $\rho_r M_r$ for all $(m_r - 1)$ possible values of ρ_r, excluding the case where $\rho_r = 0$, in each channel r; for $1 \le r \le n$.

Then, each term of the sum $Z_T = \sum \rho_r M_r$ in the CRT (Eq. (5)) is in the mixed-radix form $\langle t_{(r,n)}, t_{(r,n-1)}, \ldots, t_{(r,1)} \rangle$ with digits $t_{(r,k)}$:

$$\begin{aligned} \rho_r M_r \ = \ & t_{(r,n)} \cdot (m_{n-1} \cdot m_{n-2} \cdots m_2 \cdot m_1) \\ & + t_{(r,n-1)} \cdot (m_{n-2} \cdot m_{n-3} \cdots m_2 \cdot m_1) \\ & + \cdots \\ & + t_{(r,3)} \cdot (m_2 \cdot m_1) + t_{(r,2)} \cdot m_1 + t_{(r,1)} \end{aligned}$$
$$\text{where } 0 \le t_{(r,k)} < m_k. \tag{42}$$

This is the usual/standard mixed-radix representation associated with any given RNS; wherein, the component modulus m_n (which is the highest in magnitude among all moduli) is left out; and the channels are assigned remaining moduli arranged in a descending order of magnitude. In the above equation, note that instead of increasing powers of a single value (called the radix or base); the weights of each position include successively higher number of moduli multiplied together. Hence the name "mixed-radix" representation. Since $\rho_r M_r < M$, dividing both sides of this inequality by the product $(m_{(n-1)} \cdots m_1)$, it follows that the most significant mixed-radix digit must satisfy the constraint $t_{(r,n)} < m_n$; and the remainder must satisfy the relation Rem $< (m_{(n-1)} \cdots m_1)$. Iteratively dividing the (next) remainder by the (next) appropriate product of moduli yields the constraints on all the digits. Thus, we can define the mixed-radix digits of the summation as $\langle T_n, T_{n-1}, \ldots, T_1 \rangle$, with carry-outs c_n through c_1, given by the recursive relations:

$$\begin{aligned} T_1 &= \left(\sum_r t_{(r,1)} \right) \bmod m_1 \\ c_1 &= \left\lfloor \left(\sum_r t_{(r,1)} \right) / m_1 \right\rfloor \\ T_k &= \left(c_{k-1} + \sum_r t_{(r,k)} \right) \bmod m_k, \quad k > 1 \\ c_k &= \left\lfloor \left(c_{k-1} + \sum_r t_{(r,k)} \right) / m_k \right\rfloor, \quad k > 1. \end{aligned} \tag{43}$$

The following theorem presents an important result about the final carry-out:

Theorem 3. *If the sum in the CRT (as per Eq. (5)) is carried out in the mixed-radix format, substituting the values of $(\rho_r \cdot M_r)$ in the mixed-radix format (as per Eq. (42)), then the final carry-out c_n resulting from the summation is equal to $\mathcal{R}_C$.*

Proof. Since $Z = Z_T \mod M$, the mixed-radix form of Z is the mixed-radix sum of $\langle t_{(r,n)}, t_{(r,n-1)}, \ldots, t_{(r,1)} \rangle$ over all r.

Note that since $Z_T = Z + \mathcal{R}_C M$, and the weight of a single/unit carry-out (i.e., the numerical value of a carry-out c_n of value 1) of the Mixed-radix-format is $\mathcal{M}$, the final carry out c_n of the mixed-radix sum represents the value $c_n \cdot \mathcal{M}$. Then equating the two values of Z, it follows that $c_n = \mathcal{R}_C$. □

5.1. A carry-look-ahead framework

Our algorithm uses a carry-look-ahead technique to determine c_n. The derivation of this look-ahead framework is explained in the remainder of this sub-section. (Therefore, a casual reader can skip the rest of this section without loss of generality or missing out main concepts/ideas.)

To start with, note that each channel r can perform a summation (of the form shown in (43)); corresponding to its mixed radix-modulus m_r; in parallel.

Such an in-channel summation of small values (the mixed-radix digits, wherein, each digit or summand in channel r is strictly less than m_r, the incremental modulus for channel r) can be further sped-up because the mixed-radix digits turn out to be sparse; thereby limiting the maximum value the carry-out c_r from channel r can assume.

The following lemma shows the sparseness of the mixed-radix digits $t_{(r,k)}$.

Lemma 3. *The mixed-radix digit $t_{(r,k)}$ equals 0 for $r > k$.*

Proof. If $k = 1$, then

$t_{(r,k)} = t_{(r,1)} = \frac{\rho_r M}{m_r} \mod m_1 = 0$ when $r > 1$, since m_1 is a factor of M/m_r when $r \neq 1$.

Similarly, if $1 < k < n$, then

$$t_{(r,k)} = \left\lfloor \frac{\rho_r M}{m_r (m_1 \cdots m_{(k-1)})} \right\rfloor \mod m_k = 0 \quad \text{when } r > k. \quad \square$$

Because of the sparseness of the mixed-radix digits, we can limit the maximum value of the carries, as given in the following theorem.

Theorem 4. *The mixed-radix sum carry c_k satisfies $c_k < k$ for all k.*

Proof. From Lemma 3, $t_{(r,1)} = 0$ for $r > 1$. Thus, since $t_{(1,1)} < m_1$, we have $T_1 = t_{(1,1)}$ and $c_1 = 0$.

Similarly, for $k > 1$, $t_{(r,k)} = 0$ for $r > k$. This gives, $\sum_r t_{(r,k)} \leq k(m_k - 1)$. Consequently, from Eq. (43), $c_k \leq \lfloor (c_{k-1} + k(m_k - 1))/m_k \rfloor < k$ if $c_{k-1} < k - 1$. By induction, since $c_1 = 0 < 1$, $c_k < k$ for all k. □

Thus, by choosing $m_k \geq k$ (by ordering the radices appropriately), we can limit c_k to be less than m_k. Initially, the $t_{(r,k)}$ values can be added in each channel k in parallel (over all values of the first index r); using the h/w tree-adder in each channel; producing a sum of magnitude less than $[\lg m_k + \lg k]$ bits. By ignoring the incoming carry-ins, we can determine an approximate carry-out and sum (using Barrett reduction or a small look-up-table or any other method of choice):

$$\begin{aligned} \widehat{T_k} &= \left(\sum_r t_{(r,k)} \right) \bmod m_k \\ \widehat{c_k} &= \left\lfloor \left(\sum_r t_{(r,k)} \right) / m_k \right\rfloor < k. \end{aligned} \tag{44}$$

The actual carry-out satisfies, for $k > 1$,

$$c_k = \widehat{c_k} + \left\lfloor \left(\widehat{T_k} + c_{k-1} \right) / m_k \right\rfloor. \tag{45}$$

By choosing $m_k \geq k$, Theorem 4 gives $c_k = \widehat{c_k}$ or $c_k = \widehat{c_k} + 1$.

We define g_k as the difference between the approximate and actual channel carry-outs:

$$\mathbf{g_k} = \mathbf{c_k} - \widehat{\mathbf{c_k}} = \mathbf{0} \text{ or } \mathbf{1} \quad \text{for all } \mathbf{k} \geq \mathbf{1}. \tag{46}$$

Thus, Eq. (45) can be written as the following recurrence:

$$g_k = \left\lfloor \left(\widehat{T_k} + \widehat{c_{k-1}} + g_{k-1} \right) / m_k \right\rfloor \tag{47}$$

with g_0 and $\widehat{c_0}$ defined as 0.

For $i \leq j$, define $g^0_{[i,j]} = g_j$ if g_i were to equal 0, and $g^1_{[i,j]} = g_j$ if g_i were to equal 1. Note that $g^0_{[0,n]} = g_n$. This lends itself to a carry-look-ahead scheme to calculate $c_n = g_n + \widehat{c_n}$ in $\lceil \lg n \rceil$ steps/levels, by computing the bits $g^0_{[i,j]}$ and $g^1_{[i,j]}$ across blocks of channels of increasing size. This framework is similar to the conditional-sum addition (which is a technique for fast binary addition [19,23,12]).

As the base case, we have, for $0 \leq i < n$:

$$\begin{aligned} g^0_{[i,i+1]} &= \left\lfloor \left(\widehat{T_{i+1}} + \widehat{c_i} \right) / m_{i+1} \right\rfloor \\ g^1_{[i,i+1]} &= \left\lfloor \left(\widehat{T_{i+1}} + \widehat{c_i} + 1 \right) / m_{i+1} \right\rfloor. \end{aligned} \tag{48}$$

Note that channel k can compute $g^0_{[k-1,k]}$ and $g^1_{[k-1,k]}$ after obtaining $\widehat{c_{k-1}}$ from channel $(k-1)$.

In the second level of the carry-look-ahead tree, define for even j:

$$\begin{aligned} g^0_{[j,j+2]} &= g^0_{[j+1,j+2]} \quad \text{if } g^0_{[j,j+1]} = 0 \\ g^0_{[j,j+2]} &= g^1_{[j+1,j+2]} \quad \text{if } g^0_{[j,j+1]} = 1 \\ g^1_{[j,j+2]} &= g^0_{[j+1,j+2]} \quad \text{if } g^1_{[j,j+1]} = 0 \\ g^1_{[j,j+2]} &= g^1_{[j+1,j+2]} \quad \text{if } g^1_{[j,j+1]} = 1. \end{aligned} \tag{49}$$

In general, we have at the $(\lambda+1)$th level of the carry-look-ahead tree, for $j \bmod 2^{\lambda} \equiv 0$, and $r = 2^{\lambda-1}$:

$$\begin{aligned} g^0_{[j,j+2r]} &= g^0_{[j+r,j+2r]} \quad \text{if } g^0_{[j,j+r]} = 0 \\ g^0_{[j,j+2r]} &= g^1_{[j+r,j+2r]} \quad \text{if } g^0_{[j,j+r]} = 1 \\ g^1_{[j,j+2r]} &= g^0_{[j+r,j+2r]} \quad \text{if } g^1_{[j,j+r]} = 0 \\ g^1_{[j,j+2r]} &= g^1_{[j+r,j+2r]} \quad \text{if } g^1_{[j,j+r]} = 1. \end{aligned} \tag{50}$$

The following theorem proves that the final carry-out can easily be determined by the final value $g^0_{[0,n]}$ of the carry-look-ahead scheme.

Theorem 5. *The final carry-out c_n equals $\widehat{c_n} + g^0_{[0,n]}$.*

Proof. Without loss of generality, assume n is a power of two. If n is not a power of two, dummy g variables (with value 0) can be appended on the right side of the look-ahead tree. Lemma 6 in Appendix A states, with $\lambda = \lg n + 1$ and $r = n$, that if $g_0 = 0$, then $g_n = g^0_{[0,n]}$. Since $g_0 = 0$ by definition, $c_n = \widehat{c_n} + g_n = \widehat{c_n} + g^0_{[0,n]}$. □

5.2. *MRPR algorithm specification*

Pseudo-code for **MRPR** is given in Algorithm 3.

5.3. *Small numerical example to illustrate MRPR*

As a small example of **MRPR**, let us again use $n = 4$ with the modulus set $m_1 = 3, m_2 = 5, m_3 = 7$, and $m_4 = 11$ and extra modulus $m_e = 2$. As before, the total modulus M = 1155, the outer-weights are $M_1 = 385, M_2 = 231, M_3 = 165$, and $M_4 = 105$, and the inner-weights are $h_1 = 1, h_2 = 1, h_3 = 2$, and $h_4 = 2$.

Once again, we will calculate the value of $\mathcal{R}_C$ for $Z = 10$, with residues $z_1 = 1, z_2 = 0, z_3 = 3$, and $z_4 = 10$. We have $\rho_1 M_1 = 385$, $\rho_2 M_2 = 0$, $\rho_3 M_3 = 990$, and $\rho_4 M_4 = 945$. Channel r looks up the mixed-radix digits $t_{r,k}$ of $\rho_r M_r$, giving $t_{1,k} = \langle 3, 4, 3, 1\rangle$, $t_{2,k} = \langle 0, 0, 0, 0\rangle$, $t_{3,k} = \langle 9, 3, 0, 0\rangle$, and $t_{4,k} = \langle 9, 0, 0, 0\rangle$.

Each channel now computes an initial sum $S_r = \sum_i t_{(i,r)}$, giving $S_1 = 1, S_2 = 3, S_3 = 7$, and $S_4 = 21$. The channels then calculate the estimates $\widehat{c_r} = \lfloor S_r/m_r \rfloor$ and $\widehat{T_r} = S_r \bmod m_r$: $\widehat{c_1} = 0, \widehat{c_2} = 0$, $\widehat{c_3} = 1, \widehat{c_4} = 1$, and $\widehat{T_1} = 1, \widehat{T_2} = 3, \widehat{T_3} = 1, \widehat{T_4} = 10$.

The base values for the carry look-ahead are: $g^0_{[0,1]} = 0, g^1_{[0,1]} = 0, g^0_{[1,2]} = 0, g^1_{[1,2]} = 0, g^0_{[2,3]} = 0, g^1_{[2,3]} = 0, g^0_{[3,4]} = 1$, and $g^1_{[3,4]} = 1$.

The 2nd-level values of the carry look-ahead are: $g^0_{[0,2]} = 0$, $g^1_{[0,2]} = 0, g^0_{[2,4]} = 1$, and $g^1_{[2,4]} = 1$.

The 3rd and final level of the carry look-ahead gives: $g^0_{[0,4]} = 1$ and $g^1_{[0,4]} = 1$.

Thus, we have $\mathcal{R}_C = \widehat{c_4} + g^0_{[0,4]} = 2$.

6. Sign detection

We now present an algorithm that uses our partial-reconstruction algorithms to efficiently determine whether or not an overflow or underflow has occurred after a RNS operation. This can equivalently be used to compare the magnitude of two RNS integers. The following two lemmas will form the basis for our sign detection algorithm.

Lemma 4. *If $Z = X + Y \bmod M$, where $0 \le X, Y < M$, then the addition operation overflowed ($X + Y \ge M$) if and only if $Z \bmod m_e \ne (X + Y) \bmod m_e$ for an extra modulus m_e with* $\gcd(M, m_e) = 1$.

Algorithm 3: MRPR

Input : An integer Z in RNS form: $\overline{\mathbb{Z}} = [z_1, z_2, \ldots, z_n]$
Output: The correct value of $\mathcal{R}_C$

begin
  /* Per-channel memory stores pre-computed mixed-radix representations of $\rho_r M_r$ for all possible z_r and are looked up in parallel across channels. */
  $\langle t_{(r,n)}, t_{(r,n-1)}, \ldots, t_{(r,1)} \rangle \leftarrow MixedRadix(\rho_r M_r)$;
  /* In parallel, each channel fills the rows of a central FRiP-SCiP block in parallel as described in Section 8.2 and shown in Figure 5. Then, in parallel, each column in the block then computes an initial sum using a dedicated tree-adder, returning the sum S_r to channel r. */
  $S_r \leftarrow \left(\sum_i t_{(i,r)}\right)$;
  /* Each channel calculates an initial carry-out estimate, using Barrett reduction or a small lookup-table */
  $\widehat{c_r} \leftarrow S_r / m_r$;
  $\widehat{T_r} \leftarrow S_r \bmod m_r$;
  /* In parallel, each channel adds incoming carry and calculates base g values. */
  $V_r \leftarrow \widehat{T_r} + \widehat{c_{r-1}}$;
  if $V_r \ge m_r$ **then**
    $g^0_{[r-1,r]} \leftarrow 1$;
    $g^1_{[r-1,r]} \leftarrow 1$;
  else
    $g^0_{[r-1,r]} \leftarrow 0$;
    if $V_r + 1 \ge m_r$ **then**
      $g^1_{[r-1,r]} \leftarrow 1$;
    else
      $g^1_{[r-1,r]} \leftarrow 0$;
  /* All g values are limited to {0,1} and are processed by the carry-look-ahead block. */
  for $\lambda \leftarrow 2$ **to** $\lceil \lg n \rceil + 1$ **do**
    $R \leftarrow 2^{\lambda-1}$;
    $u \leftarrow g^0_{[j,j+R]}$;
    $v \leftarrow g^1_{[j,j+R]}$;
    $g^0_{[j,j+2R]} \leftarrow g^u_{[j+R,j+2R]}$;
    $g^1_{[j,j+2R]} \leftarrow g^v_{[j+R,j+2R]}$;
  return $(\widehat{c_n} + g^0_{[0,n]})$;

Proof. If the addition operation did not overflow, then $X + Y < \mathrm{M}$ and $Z = X + Y$. Therefore, $Z \bmod m_e = (X + Y) \bmod m_e$. Conversely, if the addition operation did overflow, then $X + Y \ge \mathrm{M}$ and $Z = X + Y - \mathrm{M}$. Therefore, $Z \bmod m_e = (X + Y - \mathrm{M}) \bmod m_e \ne (X + Y) \bmod m_e$, since $\gcd(\mathrm{M}, m_e) = 1$ and $\mathrm{M} \bmod m_e \ne 0$. □

Lemma 5. *If $Z = X - Y \bmod M$, where $0 \le X, Y < M$, then the subtraction operation under-flowed ($X - Y < 0$) if and only if $Z \bmod m_e \ne (X - Y) \bmod m_e$ for an extra modulus m_e with* $\gcd(M, m_e) = 1$.

Proof. If the subtraction operation did not underflow, then $X - Y \ge 0$ and $Z = X - Y$. Therefore, $Z \bmod m_e = (X - Y) \bmod m_e$. Conversely, if the subtraction operation did underflow, then $X - Y < 0$ and $Z = X - Y + \mathrm{M}$. Therefore, $Z \bmod m_e = (X - Y + \mathrm{M}) \bmod m_e \ne (X - Y) \bmod m_e$, since $\gcd(\mathrm{M}, m_e) = 1$ and $\mathrm{M} \bmod m_e \ne 0$. □

Consequently, the algorithm focuses on calculating Z mod m_e, for extra modulus m_e with gcd(M, m_e) = 1. The actual value of (Z mod m_e) can then be compared to an in-channel extra-modulus value: $(x+y)$ mod m_e or $(x-y)$ mod m_e. If the values are the same, then no underflow/overflow occurred.

6.1. Sign detection algorithm specification

Algorithm 4 gives an efficient sign detection method using Lemmas 4 and 5 and the partial reconstruction algorithms presented in the previous sections.

Algorithm 4: RNS Sign Detection

Input : An integer Z in RNS form: $\overline{\mathbb{Z}} = [z_1, z_2, \ldots, z_n], z_e$; extra modulus residue z_e may be invalid (due to underflow/overflow)

Output: A binary flag OverUnder; if OverUnder = 1, there was an overflow or underflow ; otherwise, OverUnder = 0

```
1 begin
    /* Call either of the partial-reconstruction
       algorithms (such as RPPR) to determine the exact
       value of R_C                                    */
2   R_C ← RPPR(Z̄) or MRPR(Z̄);
3   S_e ← (Σ ρ_r M_r mod m_e − R_C M mod m_e) mod m_e
       /* Precomputed memory stores ρ_r M_r mod m_e    */
4   if S_e == z_e then
5     return 0
6   else
7     return 1
```

6.2. The extra modulus m_e

Both **RPPR_base** and **RPPR**; as well as the sign detection algorithm, require extra information in the form of residues/remainders w.r.t. an additional modulus m_e. We therefore propose a redundant channel, or extra modulus m_e, be added to the original moduli set. All simple operations in the RNS (addition, subtraction, multiplication) should be carried out in the redundant channel in addition to the original channels.

The restrictions on m_e are

$$\gcd(M, m_e) = 1 \quad \text{and;} \quad \text{for RPPR,}$$
$$(\Phi \bmod m_e) \equiv 2 \bmod m_e. \tag{51}$$

To satisfy all requirements, we propose using

$$\text{ODD } m_r, \tag{52}$$
$$m_e = 2, \quad \text{and} \tag{53}$$
$$\text{EVEN } \Phi > 2 \tag{54}$$

so that the iterative **RPPR** multiplication factor $(\Phi - 1) \neq 1$ and $\Phi \bmod m_e \equiv 2$. By choosing $m_e = 2$ and odd m_r, the extra modulus precomputations and summations for Algorithms 2 and 4 are only one bit per entry.

7. Moduli selection

For each of the partial reconstruction algorithms presented, a channel with modulus m_r needs a look-up table with $(m_r - 1)$ entries (one entry per value of z_r) for several of the precomputed variables. This is used, for example, to look up the values of

$$\alpha_r = \lfloor 2^w \rho_r / m_r \rfloor, \text{ in Algorithm RPPR\_BASE,}$$

and

$$\langle t_{(r,k)} \rangle = \text{MixedRadix}(\rho_r M_r), \text{ in Algo. MRPR.}$$

For each of these tables, the total number of memory locations required by all moduli is

$$\triangleq \mathbb{T}_{\mathbb{M}} = \sum_{r=1}^{n} (m_r - 1) \approx \sum_{r=1}^{n} m_r. \tag{55}$$

This implies that ***in order to minimize the memory needed, each component modulus should be as small as it can be; subject to the constraint that their product must exceed the maximum value in the required range***.

Therefore, in order to cover a range $[0, R)$ we select the smallest consecutive n prime numbers starting with 3 (see Section 6.2 for why 2 is excluded), such that their product exceeds R:

$$\mathbb{M} = \{m_1, m_2, \ldots, m_n\} = \{3, 5, 7, \ldots, (n+1)\text{st prime number}\},$$
$$\text{where } \prod_{r=1}^{n} m_r = \text{M} \geq R. \tag{56}$$

For example, to cover the range $[0, 2^{1024})$, we will need 131 regular channels with $m_1 = 3$ to $m_{131} = 743$, with an extra channel for $m_e = 2$. Likewise, to cover the range $[0, 2^{2048})$, we will use 233 regular channels with $m_1 = 3$ to $m_{233} = 1481$. Finally, to cover the range $[0, 2^{4096})$, we will need 418 regular channels with $m_1 = 3$ to $m_{418} = 2897$. Thus, we can easily cover large cryptographic word lengths in under 500 channels, very common in modern many-core systems such as GPUs.

The selection of moduli suggested leads to the following two highly desirable attributes that can be analytically derived as follows:

1. The nth prime number and its index n are related by the well-known prime-counting function [31] defined as

$$\pi(x) = \text{the number of prime numbers } \leq x \approx \frac{x}{\ln x} \tag{57}$$

and therefore:

$$n \approx \pi(m_n) \approx \left(\frac{m_n}{\ln m_n} \right). \tag{58}$$

It was confirmed experimentally, if $3 \leq n \leq 1000$, covering the modulus sets of interest up to a range of $[0, 2^{11282} - 1)$, then

$$0.83 \left(\frac{m_n}{\ln m_n} \right) < n < 1.22 \left(\frac{m_n}{\ln m_n} \right). \tag{59}$$

2. The overall modulus M becomes the well known primorial function [32], which for any positive integer Q is typically denoted as "$Q\#$" and denotes the product of all prime numbers $\leq Q$. In this paper, however, we use the symbol $\mathcal{P}(Q)$ to denote the primorial function which is defined as $\mathcal{P}(1) = 1$ if $Q = 1$ and as the product of all prime numbers $\leq Q$ if $Q > 1$.

   The primorial function satisfies well-known identities [32,10]

$$2^Q < \mathcal{P}(Q) < 4^Q = 2^{2Q} \quad \text{and} \tag{60}$$
$$\mathcal{P}(Q) \approx e^Q \quad \text{for large } Q. \tag{61}$$

As a result, to be able to represent N bit numbers, i.e. the range $[0, 2^N - 1)$, in the residue domain using all available prime numbers (starting with 3), the total modulus satisfies

$$2^N \approx \text{M} = \mathcal{P}(m_n)/2 \approx \exp(m_n)/2 \tag{62}$$

and therefore

$$m_n \approx \ln(2\text{M}) \approx \lceil N \cdot (\ln 2) \rceil. \tag{63}$$

Substituting this value of m_n in Eq. (58), the number of moduli n required to cover all N-bit long numbers can be approximated as:

$$n \approx \frac{N \ln 2}{\ln(N \ln 2)} \approx \frac{N}{\lg N} \approx \frac{\lg \mathrm{M}}{\lg \lg \mathrm{M}}. \tag{64}$$

It was again confirmed experimentally, if $3 \leq n \leq 1000$, then

$$0.82\left(\frac{N}{\lg N}\right) < n < 1.01\left(\frac{N}{\lg N}\right) \tag{65}$$

$$0.82\left(\frac{\lg \mathrm{M}}{\lg \lg \mathrm{M}}\right) < n < 0.99\left(\frac{\lg \mathrm{M}}{\lg \lg \mathrm{M}}\right) \tag{66}$$

$$1.9\left(\frac{N}{\lg N}\right) < m_n < 6.6\left(\frac{N}{\lg N}\right) \tag{67}$$

$$1.9\left(\frac{\lg \mathrm{M}}{\lg \lg \mathrm{M}}\right) < m_n < 6.6\left(\frac{\lg \mathrm{M}}{\lg \lg \mathrm{M}}\right). \tag{68}$$

These analytic expressions are extremely important because they imply:

1. $n < m_n \ll \mathrm{M}$, which follows from relations (58), (63) and (64).
2. ***Moreover, both the maximum-modulus*** m_n ***as well as the number of moduli*** n ***grow logarithmically with respect to*** M.

These attributes make it possible to deploy exhaustive pre-computation and lookups (for many canonical operations) because they guarantee that the total amount of memory required grows as a *low degree polynomial of the logarithm of* M (see Section 8.5 for further details).

In closing this section, we would like to point out some additional benefits of our moduli selection:

1. This selection works in general, since for any value of the range R, multiple moduli sets always exist.
2. The moduli are relatively easy to find, because prime numbers are sufficiently densely abundant, irrespective of the value of R.
3. It fully leverages the parallelism inherent in the RNS.
4. Exhaustive pre-computation and lookup is feasible (in most operations of interest) even at cryptographic word-lengths, as demonstrated above.
5. Limiting m_n and n to small values makes it more likely to fit the entire RNS processing in a single hardware module.
6. Many-core processors are common today, and each core can easily perform the integer operations on small length operands (note that the in-channel word-length $\leq \lg m_n \approx \lg \lg \mathrm{M}$).

8. Theoretical latency, communications, and memory requirements

8.1. Assumptions about implementation

The main goal of the architecture is to fully leverage the **parallelism** inherent in the RNS. Accordingly, we assume a fully-dedicated custom VLSI design. However, most of the assumptions below can be approximated in a commodity many-core system such as a GPU (each channel is a small core), as shown by the experimental results in Section 10. A schematic diagram of the architecture to execute the **RPPR** algorithm is illustrated in Fig. 4. An ideal schematic diagram of the architecture to execute the **MRPR** algorithm is illustrated in Fig. 5.

Since m_n is the largest component-modulus, Eq. (68) gives the maximum channel word-length as:

$$w_{\mathrm{CH}}(n) = \lg m_n \approx \lg \lg \mathcal{M}. \tag{69}$$

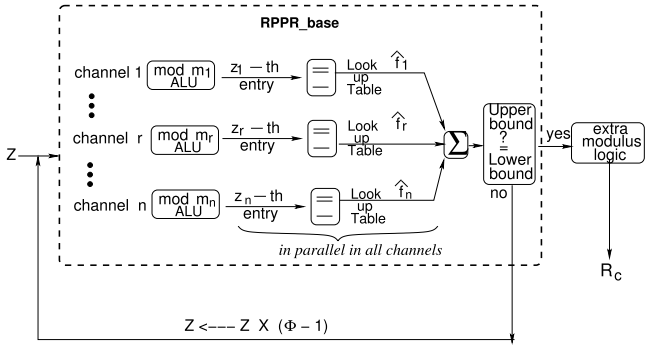


Fig. 4. A schematic of generic RPPR h/w architecture.

Note that this is drastically smaller than the wordlength w_{conv} required for conventional binary representation, which is roughly $O(N) \approx$ the number of bits required to represent $\mathcal{M}$. For instance, for $N = 4096$ bits, the maximum size of operands in any channel, viz., w_{CH} is only 12 bits.

We therefore assume each channel is capable of performing all basic arithmetic operations, viz., $\{\pm, \times, \div,$ `shifts`, `powers`, `equality-check`, `comparison`$\}$ (without any modulo or remaindering operation) as well as the operations $\{\pm, \times\}$ modulo m_r, the channel modulus. We further assume each channel is capable of accessing its own look-up-table(s) independent of other channels.

8.2. Delay assumptions, and estimation of total delay

In accordance with the literature, we make the following assumptions about delays of hardware modules:

1. A carry-look-ahead adder can add/subtract two operands within a delay that is logarithmic w.r.t. the wordlength(s) of the operands.
2. More generally, a fast-multi-operand addition of n numbers each of which is w-bits long requires a delay of
$$O(\lg n) + O(\lg(w + \lg n)) \approx O(\lg w + \lg n). \tag{70}$$
3. Assuming that the address-decoder is implemented in the form of a "tree-decoder", a look-up table with $\mathbb{L}$ entries requires $\approx O(\lg \mathbb{L})$ delay to access any of its entries.
4. Each channel has a dedicated shifter (e.g., a multi-stage-shifter also known as a "barrel" shifter [19,36]) to quickly implement shifts of variable number of bit/digit positions.
5. The routing is hard-wired to the extent possible in order to minimize wire-delays.
6. For **MRPR**, the availability of a hardware block that fills the rows of an n by n matrix in parallel and then adds all the columns in parallel. Each entry in the matrix is at most $\lg m_n$ bits. We abbreviate this "Fill Rows in Parallel" followed by "Sum Columns in Parallel" block as a ***FRiP–SCiP*** block. Each channel has precomputed mixed-radix digits of $\rho_r M_r$ for every possible z_r value. In a custom VLSI design (such as the one illustrated in Fig. 5), once this value is available, each channel sends the vector of n elements to the block, filling special purpose registers (denoted $\mathrm{R}_{k1}, \mathrm{R}_{k2}, \ldots, \mathrm{R}_{kn}$ for row k). Thus, all the rows of the "matrix" can be filled in parallel. A dedicated fast adder-tree in each column can then sum the entries of each column (in parallel with the other columns) with a delay of $O(\lg n)$.

8.3. Delay estimation

The overall delay estimation for **RPPR_base** is summarized in Table 1. The dominant delay is in Step 3, the accumulation of

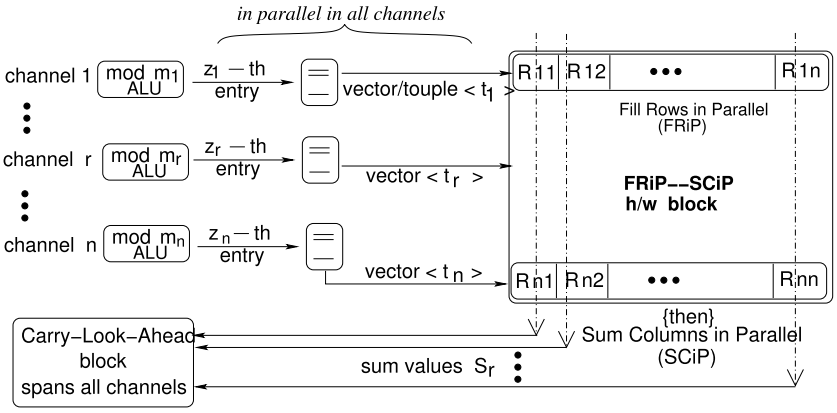


Fig. 5. Schematic of h/w architecture for **MRPR** algorithm, including the FRiP–SCiP h/w block; and the carry-look-ahead block which can be distributed across and spans all channels.

Table 1
Total delay of **RPPR_base**.

Step	Delay	Justification
1	$\lg n$	In parallel
2	$\lg n$	*Assumption* 2
3	$\lg n$	*Assumption* 2
4	$\ll \lg n$	One small addition
5	$\ll \lg n$	One small shift (can be hard-wired)
6	$\ll \lg n$	One small shift
7	$\ll \lg n$	One small comparison
Total delay $\approx O(\lg n)$		

estimate values $\widehat{f_r}$, read from the tables. Likewise, the total critical path delay of **MRPR** is summarized in Table 2.

The performance (total number of operations at run-time which is related to total power consumption; and number of operations in the critical path, which determines the critical path delay) of the **RPPR** and **MRPR** algorithms is summarized in Table 3.

8.4. Communication requirements

Since the channels are realized as distributed nodes in a many-core architecture, it is crucial for the inter-node communications to be tightly bounded. The partial reconstruction algorithms take full advantage of parallel and independent per-channel hardware and therefore have limited inter-channel communications.

For the **RPPR** iterative algorithm, we assume that a master node controls all the channels. In the first call to **RPPR_base** , the master node transfers the appropriate input RNS integer residue, as well as Φ, to each node (one communication per channel). The nodes then transfer the pre-computed approximation α_r back to the master node for a small accumulation (one communication per channel).

If the $\mathcal{R}_C$ requires disambiguation, the master node sends a small message to each node to multiply the current residue value by the small constant $(\Phi - 1)$ and return the updated approximation α_r for accumulation, as well as $(\rho_{r,Y} M_r \bmod m_e)$ value required for disambiguation.

In summary, there are $2n$ communications required per call to **RPPR_base** . Therefore, **RPPR** requires $\Theta(n \log_\Phi M)$ communications in the worst-case and $\Theta(n)$ communications on average.

In the **MRPR** algorithm, we again assume that a master node controls all the channels. The master node first transfers the appropriate input RNS integer residues to each node (one communication per channel). Each node can then in parallel lookup the mixed-radix digits for $\rho_r M_r$, with channel r sending the digits $t_{r,k}$ to the FRiP–SCiP block. Since $t_{r,k} = 0$ for $r > k$ and channel r will make use of digit $t_{r,r}$, it only sends the digits with $k > r$. This is one communication per channel, although channel r sends $n - r$ digits.

Next, channel k receives the summation of the digits $t_{r,k}$, with $k > r$, from the FRiP–SCiP block. This is also one communication per channel. Each channel can then compute $\widehat{T_r}$ and $\widehat{c_r}$, in parallel. The carry-propagation will require $n/(2^{(\lambda-1)}) - 1$ communications at the λth level, totaling less than $2n$ communications for the carry-propagation. Therefore, **MRPR** requires $\Theta(n)$ communications in all cases.

This demonstrates that the number of communication required in a distributed architecture grows as a low (first) degree polynomial in the number of channels n.

8.5. Storage requirements

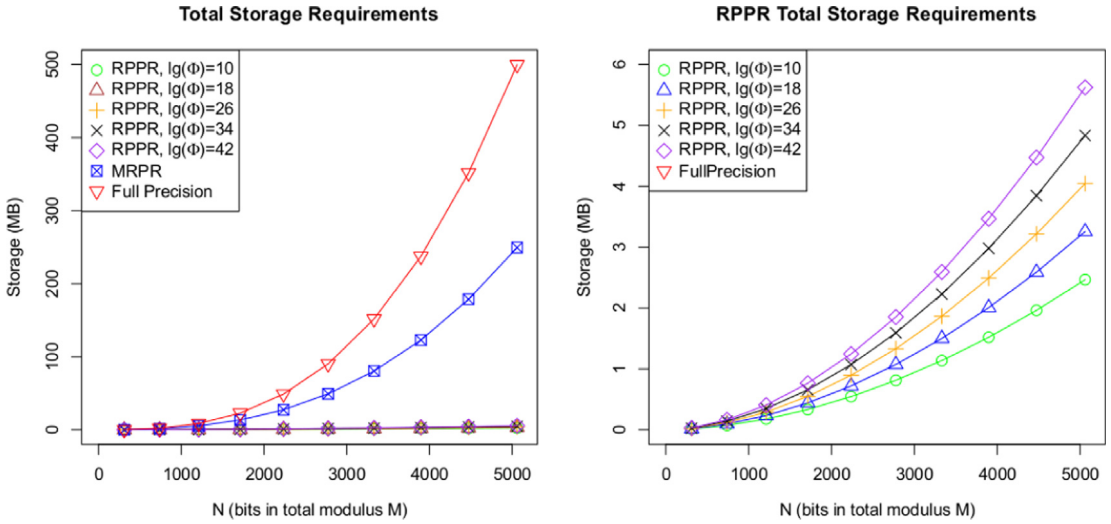
The partial reconstructions algorithms utilize a significant space–time trade-off by taking advantage of exhaustive pre-computations. By choosing the moduli in the optimal manner discussed above, the pre-computation storage requirements are shown in Fig. 6(a). The specific calculations assume (in a non-optimized fashion), that each individual storage location requires an integer multiple of bytes. As can be seen, the storage requirements grow as a low order polynomial of the overall bit-length N.

Table 2
Total delay of **MRPR**.

Step	Delay	Justification
2	$\lg n$	In parallel and *assumption* 3
3	$\lg n$	In parallel and *assumption* 2, 5 *and* 6
4	$\ll \lg n$	Two shifts and one small multiplication (Barrett reduction) in parallel
5	$\ll \lg n$	One small subtraction and multiplication (Barrett reduction) in parallel
6	$\ll \lg n$	One small add per channel, in parallel
7–15	$\ll \lg n$	Small ops in each channel in parallel
16–22	$\lg n$	⋆⋆ Carry-look-ahead-tree across all channels few ($O(1)$) bits used from each channel
Total delay $\approx O(\lg n)$		

Table 3
Operation counts of all partial reconstruction algorithms.

Algorithm	Small $O(\lg m_n)$ Multiplications		Small $O(\lg m_n)$ Additions		# calls to its base function
	Total	*Critical path*	Total	*Critical path*	
RPPR_BASE (in each call)	0	0	$\Theta(n)$	$\Theta(\lg n)$	–
RPPR (counts per call)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(\lg n)$	–
RPPR (worst case)	$\Theta(n \log_\Phi M)$	$\Theta(\log_\Phi M)$	$\Theta(n \log_\Phi M)$	$\Theta((\lg n)(\log_\Phi M))$	$\Theta(\log_\Phi M)$
RPPR (average)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(1)$
MRPR(all cases need identical operations)	$\mathbf{\Theta(n)}$	$\mathbf{\Theta(1)}$	$\mathbf{\Theta(n^2)}$	$\mathbf{\Theta(lgn)}$	**0**



(a) Total storage needed for the look-up tables.

(b) Zoomed-in view of previous sub-figure on the left.

Fig. 6. Comparison of total storage needed by **RPPR** and **MRPR** vs. existing methods (plot labeled full precision in sub-figure (a)). Note that this plot and **MRPR** are out of scale and therefore do not appear in (the next) sub-figure (b).

Fig. 6(a) clearly illustrates that all versions of **RPPR** require substantially smaller amount of storage for the precomputed look-up tables compared to storing the full precision fractions, while **MRPR** requires approximately half that of full precision. Fig. 6(b) is a zoomed-in version of the previous sub-figure, showing the differences among different Φ values for **RPPR**. The plots for the full-precision method and **MRPR** are out of scale and therefore does not appear in the zoomed-in figure.

For instance, the total number of memory locations needed by **RPPR** is

$$\#\,\text{loc} = \left(\sum_{r=1}^{n}(m_r - 1)\right) \lesssim O\left(\frac{n(n+1)}{2}\right) \approx O(n^2)$$
$$\approx O\left(\frac{N^2}{\ln^2 N}\right) \tag{71}$$

where the value of n is substituted from Relation (64).

For **MRPR** each location stores the value of $(\rho_r \cdot M_r)$ in a mixed-radix format. Therefore, the number of bits required per location is

$$N_{\text{bits}}/\text{per loc} = \sum_{r=1}^{n-1} \lceil \lg m_r \rceil \approx \lg \text{M} \approx O(N).$$

Therefore,

$$\text{total storage (in bits)} = O\left(\frac{N^2}{\lg^2 N}\right) \times O(N)$$
$$\approx O\left(\frac{N^3}{\lg^2 N}\right). \tag{72}$$

However, this can be optimized, since $t_{r,k} = 0$ for $r > k$. By not storing the digits when $r > k$, we reduce the storage requirements of **MRPR** by approximately half.

As a further optimization, we use residue values z_r directly as indexes for look-up. Note that in each channel k, the ρ_k values are obtained from z_k value by modular multiplication with the constant h_k (i.e., the inner weight in each channel k, these are constants once the set of moduli is selected). Therefore a simple permutation of the rows (of the table which is indexed by ρ_r) allows the residues z_k to be directly used as indexes into the look-up tables. We call such a table the *Residue Addressed Table* (abbreviated as *RAT*).

In closing this section, we point out that the storage requirements are far less onerous than they appear for the following reasons.

1. At run time, the storage is for look-up only. In other words, the memory required is read-only. Read-only memory cells can be much smaller, faster and power efficient than read/write memory cells.
2. The memory is not one large monolithic block wherein any location is accessible at random. Rather the memory can be distributed among all channels and "random-access" is limited to the set of $(m_r - 1)$ entries in each channel. In fact, for $\Phi \leq 42$ and $n \leq 500$, **RPPR** requires less than 25 kB of precomputed memory in the largest channel m_n. The implication is that the address decoders need to deal with small numbers and are therefore small in size and consume substantially lower power.
3. All the accesses to per-channel tables within each channel are independent of accesses in other channels and therefore all of them can be done in parallel. The implication is that the access times are likely to be a lot better than a single large block of RAM.

Table 4
Comparisons of RNS Sign Detection based on different algorithms.

Method	Attribute					
	Exhaustive look-up at cryptographic bit-lengths considered?	All operands small?	Max operand length?	Extra modulus needed?	Sign/overflow detect?	Comments
[33]	No	No	$\lg(M \cdot n) \equiv$ full-precision bit-length	No	No	From Eqn (14) & Step 3 in [33]
[8,20]	No	No	$\lg(M \cdot n)$	No	Yes	Eqn (9) & following para in [20]
[11]	No	No	$\lg(\sum_{r=1}^{n} M_r)$			Sections IV, V in [11]
[35]	No	No	$O(\lg \sqrt{M})$	No	Yes	Table 1 in [35]
[29]	No	No	$O(\lg \frac{M}{4})$	No	Yes	From Eqn (1) & Table 1 in [29]
[1]	No	No	$O(\sum_{r=1}^{n} M_r)$	No	Yes	Eqn (2) & Algorithm 3.1 in [1]
RPPR	Yes	Yes	$\lg(n \cdot \Phi)$	Yes	Yes	
MRPR	Yes	Yes	$\lg(n \cdot m_n)$	No	Yes	

9. Comparison with other fast sign-detection methods

A high-level comparison of our sign detection algorithms with other sign detection methods in the literature is given in Table 4. Brief explanatory notes about each of these other methods are as follows.

In [33,8,20], full precision lookup tables are utilized, requiring substantially more pre-computed storage. We experimentally show in Section 10, that the average run-time of **RPPR** is also substantially faster than these methods.

In [11], a pre-computed lookup table is required with $\sum M_r$ entries. This would require exponential storage, i.e. $\Theta(2^N)$ memory locations, not practical at cryptographic bit lengths.

Both [35] and [1] do not allow for general modulus sets, one of our requirements. Furthermore, they also require operations on very long operand lengths of approximately $N/2$ and N bits, respectively, which does not take full advantage of the parallelism inherent in RNS.

In [1], specific parameters of core functions must be pre-computed by exhaustively searching the RNS space, which will not scale. *Additionally, this method cannot be used for magnitude comparison or overflow/underflow detection, as the parity of Z is assumed to be available as an input to the algorithm.* As previously shown in Section 3.1, an overflow/underflow can corrupt the value of $(Z \bmod 2)$ (and therefore the exact value of $(Z \bmod 2)$ must be computed, it cannot be assumed to be available).

In prior literature, the best known method to convert a RNS value into the corresponding mixed-radix format using operations only on small (channel-length) operands is iterative (see chapter 11, section 11.3 in [19]). The iterations are completely sequential and do not make use of a sparse space of mixed-radix digits; it therefore requires $O(n^2)$ operations which results in $O(n^2)$ delay, substantially higher than **MRPR**.

10. Experimental results from a GPU implementation

The delay and area estimates in Section 8 assume a dedicated full-custom VLSI implementation.

An intermediate step before committing to a full custom VLSI design is to run the algorithms on modern GPUs with thousands of cores. Ideally, with sufficient number of cores available, each core can be dedicated to realize one RNS channel.

Another advantage of an implementation is that it can independently corroborate the correctness of the algorithms. This is true irrespective of whether the implementation is done on a many-core GPU; or using FPGAs; or in the form of a dedicated full-custom VLSI chip. This fact together with the other fact that a GPU implementation is the easiest to realize (as compared with synthesis on FPGAs or custom-VLSI design and fabrication) clearly indicated that the best way to test the algorithms was to run them on many-core GPU hardware.

Therefore, we tested our algorithms by running them on a recently introduced top of the line GPU: the Nvidia 980 TI which has 2800+ of FP32-CUDA-Cores/GPU [22] (i.e., each core has the capability to perform 32-bit floating point operations independently). The total on-chip memory available in various shared configurations is more than 6 GB. The vendor (Nvidia) has designed a software development tool named "CUDA" that parallelizes the code execution across all available cores as optimally as it can. We therefore implemented our algorithms as well as full-precision (conventional) methods in CUDA and ran it on the GPU. The results are summarized in the figures in this section.

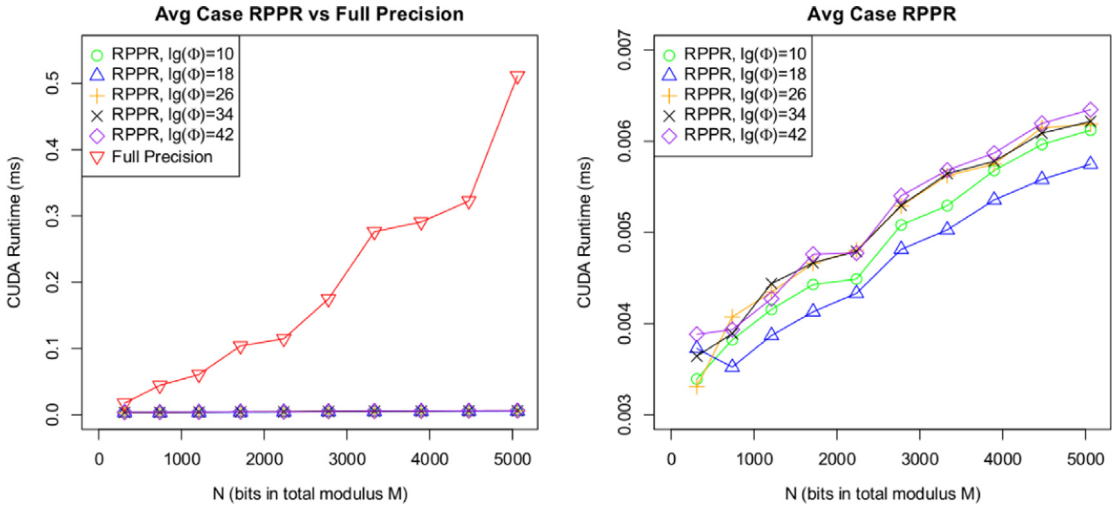
In all the figures (in which they appear), plots labeled "full precision" correspond to the delay of existing/conventional methods (such as [33,34] in Table 4), all of which require full precision computations. The full-precision delay plot is consistently drawn in bright red color and the points on the plot are indicated by the downward-pointing-triangle symbol ($\triangledown$).

For the **RPPR** algorithm, since the delay depends on the precision parameter Φ, we show distinct plots for each value of Φ. Out of this cluster of plots, the one of main interest corresponds to the largest value of Φ (which in the simulations was 42). That plot is also consistently drawn in the same (purple) color across all figures; with the points on the plot indicated by the diamond symbol ($\diamond$).

Fig. 7(a) compares the average run-time (delay) required to determine the exact value of $\mathcal{R}_C$ using the iterative RPPR algorithm (i.e., Algorithm 2 unveiled in this paper); with the delay required to do the same computation using existing methods (all of these require full precision computations). The plots demonstrate the following main points:

1: The average delay of **RPPR** algorithm is substantially lower than the delay of existing methods for all cryptographic word-lengths in a wide range from 512 bits all the way up through 5000+ bits.
2: In the large vertical scale needed to accommodate the plot showing the delay of other (full precision) methods; delays of **RPPR** methods (with different values of the precision parameter Φ) are so close to each other that they appear to coincide into a single plot (a flat line near the x-axis).
3: The fact that this cluster-plot is almost flat (horizontal) corroborates Theorem 7 (in Appendix A) which states that the expected (or average) number of iterations of **RPPR** is $\Theta(1)$ and *independent of N*.

Fig. 7(b) shows a zoomed-in view that separates all the clustered **RPPR** plots. Note that the full-precision plot is out of scale and therefore does not appear in this figure.



(a) Execution delays of **RPPR** vs. existing methods.

(b) Zoomed in view of previous sub-figure on the left.

Fig. 7. Comparison of average delays of RPPR vs. existing methods (all existing methods need full precision and are therefore represented by the plot labeled full-precision in sub-figure (a) above). Note that this plot is out of scale and therefore does not appear in (the next) sub-figure (b).

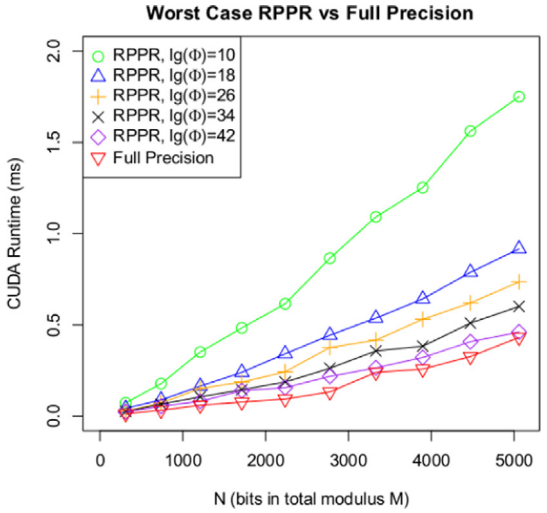


Fig. 8. Comparison of worst case delays of RPPR vs. other existing methods (plot lebeled full precision).

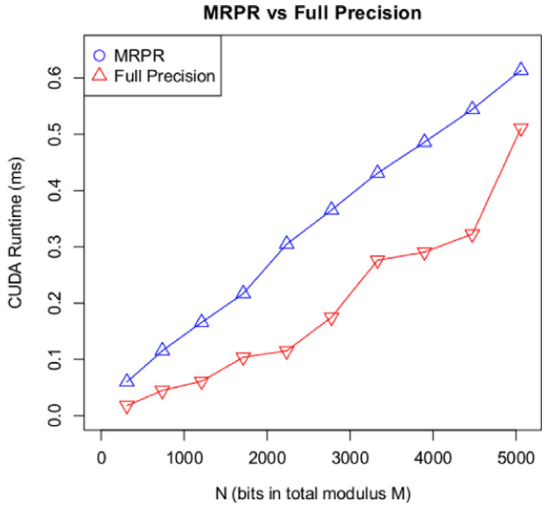


Fig. 9. Comparison of the delay of full-precision methods vs. the mixed-radix or M **RPPR**.

Fig. 8 compares the delay of the iterative version of **RPPR** in the worst case which happens to be at $Z = 1$. In this case, the delay of full-precision methods is comparable to the delay of **RPPR** with $\Phi = 42$.

However, it turns out that the probability of encountering this and/or other worst cases in real-life RSA encryptions and decryptions is vanishingly small. This is the case because of the following reasons:

1. Let $\mathcal{N} = P \times Q$ be the publicly known modulus of the RSA system. For high security, the primes P and Q are carefully chosen as described in [14].
2. Even if the actual plain-text has specific values (such as 0 or 1 etc.). There are padding procedures (ex: see [26]) which ensure that the padded value, which is the input to the encryption method is sufficiently large and appears randomly distributed.
3. From the above two constraints it can be shown that the probability of landing upon intermediate remainder $Z = 1$ (which is the worst case for **RPPR**); is negligible (the probability $= 0$ in a measure theoretic sense as well as a practical sense). More generally the probabilities of encountering very small positive or negative values of intermediate remainders (these are the bad cases for **RPPR** since these values lie deep/far outside the No-Ambiguity-Zone) are vanishingly small.

The observation that bad cases for **RPPR** are extremely rare is consistent with the fact that the *average* number of iterations of **RPPR** is $\Theta(1)$ (see Theorem 7 in Appendix A).

In summary the experimental data demonstrates that the iterative **RPPR** algorithm provides a better performance, i.e., smaller execution delay while simultaneously needing a substantially smaller storage.

The last figure in this section (i.e. Fig. 9) illustrates the results from our implementation of **MRPR** (Algorithm 3 in Section 5) in CUDA.

The implementation confirmed that the algorithm and the carry-lookahead method compute the correct results. Further, because of the upper triangular nature of the matrix (of operands, wherein all elements in each column need to added); the storage needed by **MRPR** turns out to about half of what is required by Full-Precision methods.

On the down-side, the plots appear to indicate that **MRPR** ran a bit slower than full-precision methods in CUDA.

This is an artifact caused by the limitations of shared-memory modules in many-core GPUs. These limitations require that the

$O(n^2)$ bit/digit operations needed to "fill" the $n \times n$ matrix cannot be done entirely in parallel. A serialization must be introduced in the filling of one of the two dimensions (either the rows or the columns). This serialization in turn causes the delay of the matrix-filling operation to be $O(n)$. The overall delay is therefore $O(n)$ as opposed to $O(\lg n)$ as claimed in Table 4.

This fact does not contradict the claim in Table 4. It does, however, clearly bring out the distinction between the capabilities of a many-core implementation vs. a full-custom VLSI realization. A full custom hardware implementation can easily afford all the blocks shown in Fig. 5. The delay of each block including the ***FRiP–SCiP*** block as well as the carry-look-ahead logic was estimated to be no bigger than $O(\lg n)$ in Section 8.2. In other words, a many-core implementation cannot achieve what the cascade of ***FRiP–SCiP*** block followed by the carry-look-ahead logic can in a full custom VLSI implementation (or in an FPGA realization).

11. Conclusion

We have identified that computing the exact value of the *reconstruction-coefficient* ($\mathcal{R}_C$) is the main bottleneck that makes it hard to implement fast sign (or overflow/underflow) detection. We showed two different highly-parallel methods to efficiently evaluate the $\mathcal{R}_C$, using limited inter-channel communications, and also illustrating a wide range of possibilities/trade-offs between memory (to store precomputed results and look those up at runtime) vs. performing the computations at run-time.

In effect, we have shown that all of the following problems:

1. Exact determination of $\mathcal{R}_C$
2. Overflow/underflow detection
3. Sign detection
4. Magnitude comparison
5. Determining $(Z \bmod 2)$ when $\gcd(M, 2) = 1$

are equivalent in RNS in the sense that if any one of the above problems can be solved efficiently, then all other problems can also be solved efficiently.

Thus, we have cleared away an entire log-jam of RNS operations (listed immediately above) that were hitherto considered difficult to expedite; by

(i) *identifying their equivalence and*
(ii) *solving the first problem in that equivalence-class (viz., ultra-fast determination of $\mathcal{R}_C$).*

Another independent contribution of this work is the fact that (for many canonical arithmetic operations of interest, including all the operations considered in this paper); our methods show how to exhaustively cover all possible input cases, even at asymptotically large operand bit-lengths; without incurring a super-polynomial growth in the size of the storage required.

Follow-on work includes the incorporation of the partial reconstruction algorithms and exploration of the FRiP–SCiP block in other operations including base change, scaling, modular reduction and modular exponentiation and evaluation of their performance.

Acknowledgments

The authors would like to thank their colleague Prof. Ryan Robucci for making available GPU hardware and the associated CUDA software on high end graphics machine(s) in the Eclipse Lab cluster (which he co-directs with other faculty members in the CSEE Dept. at UMBC, for more info please visit http://eclipse.umbc.edu).

We would also like to acknowledge the constructive comments and suggestions from the anonymous reviewers of the manuscript.

Appendix A. Additional proofs

Theorem 6. *If λ is the final value of* `Iter` *in Algorithm 2 (**RPPR**), then $\lambda \le \lceil \log_{(\Phi-1)} M \rceil$.*

Proof. Let W_i denote the RNS number with residues w_r after the ith iteration of the while loop, where $W_i = (\Phi-1)^i Z \bmod M$. Due to Theorem 2, when $\frac{M}{\Phi} \le W_i \le \frac{M(\Phi-1)}{\Phi}$, the while loop will stop with $\lambda = i+1$. Otherwise, there are two cases: (a) $W_i = X$ or (b) $W_i = M - X$, where $0 < X < \frac{M}{\Phi}$. Let $W_{i+1} = (\Phi-1)W_i \bmod M$.

For case (a), $W_{i+1} = (\Phi-1)X$, with $0 < W_{i+1} < \frac{M(\Phi-1)}{\Phi}$. In other words, multiplying by $\Phi - 1$ brings W_i closer to the NDZ (by a factor of $\Phi - 1$), but does not overshoot the NDZ. In the worst case, when $Z = 1$, $(\Phi-1)^i \ge \frac{M}{\Phi}$ when $i = \left\lceil \frac{\log M - \log \Phi}{\log(\Phi-1)} \right\rceil$.

For case (b), $W_{i+1} = (\Phi-1)(M-X) \bmod M = M - (\Phi-1)X$, with $\frac{M}{\Phi} \le W_{i+1} < M$. Similarly, multiplying by $\Phi - 1$ brings W_i closer to the NDZ (by a factor of $\Phi - 1$), but does not overshoot the NDZ. In the worst case, when $Z = M-1$, $(\Phi-1)^i(M-1) \bmod M \le \frac{M(\Phi-1)}{\Phi}$, or $-(\Phi-1)^i \le -\frac{M}{\Phi}$, when $i = \left\lceil \frac{\log M - \log \Phi}{\log(\Phi-1)} \right\rceil$.

In all cases, $\lambda = i_{\max} + 1 \le \left\lceil \frac{\log M - \log \Phi}{\log(\Phi-1)} \right\rceil + 1 \le \lceil \log_{(\Phi-1)} M \rceil$. □

Theorem 7. *For Z randomly chosen uniformly in $0 < Z < M$, if λ is the final value of* `Iter` *in Algorithm 2 (**RPPR**), then the expected value of λ is $\Theta(1)$. Specifically, $E[\lambda] < \frac{\Phi}{\Phi-2}$.*

Proof. Let W_i denote the RNS number with residues w_r after the ith iteration of the while loop, where $W_i = (\Phi-1)^i Z \bmod M$. Following the proof of Theorem 6, there are two cases when W_i is not in the NDZ: (a) $W_i = X$ or (b) $W_i = M - X$, where $0 < X < \frac{M}{\Phi}$.

For case (a), $W_{i+1} = (\Phi-1)X$, with W_{i+1} guaranteed to lie in the NDZ when $W_{i+1} \ge \frac{M}{\Phi}$. This happens if $X \ge \frac{M}{\Phi(\Phi-1)}$; thus, for at least $\frac{\Phi-2}{\Phi-1}$ of the uniformly chosen X, W_{i+1} will lie in the NDZ.

For case (b), $W_{i+1} = (\Phi-1)(M-X) \bmod M = M - (\Phi-1)X$, with W_{i+1} guaranteed to lie in the NDZ when $W_{i+1} \le M - \frac{M}{\Phi}$. Similarly, this happens if $X \ge \frac{M}{\Phi(\Phi-1)}$, and for at least $\frac{\Phi-2}{\Phi-1}$ of the uniformly chosen X, W_{i+1} will lie in the NDZ.

Note that Algorithm 2 is guaranteed to not enter the while loop ($\lambda = 1$) when $\frac{\Phi}{M} \le Z \le \frac{M(\Phi-1)}{\Phi}$. Thus, at least $\frac{\Phi-2}{\Phi}$ of the uniformly chosen Z will have $\lambda = 1$.

Putting this all together,

the expected value of λ is at most

$$
\begin{aligned}
&= \frac{\Phi-2}{\Phi} + 2\left(\frac{2}{\Phi}\right)\left(\frac{\Phi-2}{\Phi-1}\right) + 3\left(\frac{2}{\Phi}\right)\left(\frac{\Phi-2}{(\Phi-1)^2}\right) + \cdots \\
&\quad + \lambda_{\max}\left(\frac{2}{\Phi}\right)\left(\frac{\Phi-2}{(\Phi-1)^{(\lambda_{\max}-1)}}\right) \\
&= 2\left(\frac{\Phi-2}{\Phi}\right)\left(\frac{1}{2} + \frac{2}{(\Phi-1)} + \frac{3}{(\Phi-1)^2}\right. \\
&\quad \left. + \cdots + \frac{\lambda_{\max}}{(\Phi-1)^{(\lambda_{\max}-1)}}\right). \qquad (73)
\end{aligned}
$$

Now the infinite series

$$
\begin{aligned}
\sum_{i=1}^{\infty}\left(\frac{i}{(\Phi-1)^{i-1}}\right) &= \left(\frac{\Phi-1}{\Phi-2}\right)\sum_{i=1}^{\infty}\left(\frac{1}{(\Phi-1)^{i-1}}\right) \\
&= \left(\frac{\Phi-1}{\Phi-2}\right)^2 \quad \text{when } \Phi > 2.
\end{aligned}
$$

Therefore, $E[\lambda] < 2\left(\frac{\Phi-2}{\Phi}\right)\left(\frac{\Phi-1}{\Phi-2}\right)^2 - \left(\frac{\Phi-2}{\Phi}\right) = \frac{\Phi^2-2}{\Phi^2-2\Phi} < \frac{\Phi}{\Phi-2}$ when $\Phi > 2$. □

Table 5
The Residue Addressed Table (**RAT**) for the **RPPR** algorithm (2048 bit operands).

Channel ⟨ No. , modulus m_r⟩ ↓	Table entries (≡ ordered pairs) for row with modulus m_r : **entry in column i** ⟵ $\left(\left\lfloor \frac{\rho_i \times C_s}{m_r} \right\rfloor , (\rho_i \times \mathcal{M}_r) \bmod m_e\right)$ wherein, $\rho_i = ((i \times h_r) \mod m_r)$; and C_s = scale-factor ⟵ column index i ⟶					
	1	2	⋯	738	⋯	1480
⟨1, 3⟩ ⟶	(666, 0)	(333, 1)				
⟨2, 5⟩ ⟶	(200, 1)	(400, 0)	(600, 1)	(800, 0)		
⋮			⋯ ⋮ ⋯			
⟨130, 739⟩ ⟶	(870, 1)	(740, 1)	⋯	(129, 0)		
⋮				⋯ ⋮ ⋯		
⟨233, 1481⟩ ⟶	(781, 1)	(562, 1)	⋯			(218, 0)

Lemma 6. *Let* $r = 2^{\lambda-1}$. *In the* ***MRPR*** *method, if* $g_j = 0$, *then* $g_{(j+r)} = g^0_{[j,j+r]}$. *Otherwise, if* $g_j = 1$, *then* $g_{(j+r)} = g^1_{[j,j+r]}$.

Proof. Assume the lemma is true for $\lambda = \lambda_0$. We will show it is then true for $\lambda = \lambda_0 + 1$. Let $r_0 = 2^{\lambda_0 - 1}$.

By our assumption, if $g_j = 0$, then $g_{(j+r_0)} = g^0_{[j,j+r_0]}$, and if $g_j = 1$, then $g_{(j+r_0)} = g^1_{[j,j+r_0]}$. Further, if $g_{(j+r_0)} = 0$, then $g_{(j+2r_0)} = g^0_{[j+r_0,j+2r_0]}$, and if $g_{(j+r_0)} = 1$, then $g_{(j+2r_0)} = g^1_{[j+r_0,j+2r_0]}$. We have four possibilities to consider.

First, if $g_j = 0$ and $g_{(j+r_0)} = 0$, then $g^0_{[j,j+r_0]} = 0$ and $g^0_{[j+r_0,j+2r_0]} = g_{(j+2r_0)}$. Using the first relation in Eq. (50), $g^0_{[j,j+2r_0]} = g_{(j+2r_0)}$.

Second, if $g_j = 0$ and $g_{(j+r_0)} = 1$, then $g^0_{[j,j+r_0]} = 1$ and $g^1_{[j+r_0,j+2r_0]} = g_{(j+2r_0)}$. Using the second relation in Eq. (50), $g^0_{[j,j+2r_0]} = g_{(j+2r_0)}$.

Third, if $g_j = 1$ and $g_{(j+r_0)} = 0$, then $g^1_{[j,j+r_0]} = 0$ and $g^0_{[j+r_0,j+2r_0]} = g_{(j+2r_0)}$. Using the third relation in Eq. (50), $g^1_{[j,j+2r_0]} = g_{(j+2r_0)}$.

Fourth, if $g_j = 1$ and $g_{(j+r_0)} = 1$, then $g^1_{[j,j+r_0]} = 1$ and $g^1_{[j+r_0,j+2r_0]} = g_{(j+2r_0)}$. Using the fourth relation in Eq. (50), $g^1_{[j,j+2r_0]} = g_{(j+2r_0)}$.

Therefore, if $g_j = 0$, then $g_{(j+2r_0)} = g^0_{[j,j+2r_0]}$, and if $g_j = 1$, then $g_{(j+2r_0)} = g^1_{[j,j+2r_0]}$. Thus, the lemma is true for $\lambda_0 + 1$.

Finally, for the inductive base case, $\lambda = 1$ and $r = 1$, and the lemma is true based on the definition of $g^0_{[k-1,k]}$ and $g^1_{[k-1,k]}$ given in Eq. (48). □

Appendix B. Sample Residue Addressed Table (RAT)

Since exhaustive pre-computation and look-up is at the core of the **RPPR** methods, we show a sample Residue-Addressed-Table. Before we do that, we outline the moduli selection procedure.

B.1. Moduli selection

Throughout this manuscript we have considered a RNS to implement a 1024 bit RSA cryptosystem. We select the extra-modulus $m_e = 2$ as stated required by conditions (51) at the very end of Section 6.

(a) Single Precision calculations ⇒ 1024 bit operands, Range $\mathcal{R}_S = (2^{(1024)} - 1)$: $\mathbb{M}_S = \{3, 5, 7, 11, 13, 17, 19, \ldots, 743\}$; $n_S = |\mathbb{M}_S| = 131$; and $\mathcal{M}_S = 3 \times 5 \times 7 \times \cdots \times 743$.

(b) Double Precision calculations ⇒ 2048 bit operands (produced for example by squaring a 1024 bit integer), Range $\mathcal{R}_D = (2^{(2048)} - 1)$: $\mathbb{M}_D = \{3, 5, 7, 11, 13, 17, \ldots, 1481\}$; $n_D = |\mathbb{M}_D| = 233$; and $\mathcal{M}_D = 3 \times 5 \times 7 \times \cdots \times 1481$.

B.2. Selection of precision parameter Φ

For the ease of illustration we pick the smallest allowable value that satisfies conditions (51) ⇒ $\Phi = 4$.

Then using Theorem 2, we find the required number of fractional decimal digits of precision for the RAT table: $w_F = \lceil \log_{10}(4 \times 233) \rceil = 3$ ⇒ the scaling-factor (for integer-only realization) is $C_s = 10^3 = 1000$.

B.3. The Residue Addressed Table (RAT) for RPPR

This appendix shows the RAT used by the **RPPR** algorithm.

First, note that each row of Table 5 is a (sub) table corresponding to one modulus.

For example, the 130th modulus $= m_{130} = 739$. Accordingly, the row corresponding to 739 is the table that is included in the 130th channel. For this channel, $0 \le z_{130} \le 738$, and therefore that row/sub-table has 738 entries corresponding to each of the non-zero values that the residue z_{130} can assume for the 130th channel (one listed under each column). As explained above, each entry is an ordered pair, wherein, the first element is the approximate fractional estimate that is pre-computed, truncated to 3 decimal places and multiplied by a scaling factor 10^3 to convert it into an integer. As a result the first element of the ordered pair needs at most 3 decimal digits.

The second element is a modulo-2 value, which is sent to the extra channel for accumulation modulo-2. This computation is necessary if iterative disambiguation is required.

To conclude the table description we explain how some of the entries are calculated. Let us consider the modulus 130th channel with modulus $m_{130} = 739$ again. In this case, it can be verified that the inner-weight (defined in Eq. (8)) for this channel is $h_{130} = 643$; and $(\mathcal{M}_{130} \bmod 2) = 1$.

As a result, for $z_{130} = [1, 2, \ldots, 738]$; the corresponding reconstruction remainders are $\rho_{130} = [643, 547, \ldots, 96]$; the fractions are

$$f_r = \left[\frac{643}{739}, \frac{547}{739}, \ldots, \frac{96}{739}\right].$$

The approximate values truncated to 3 digits are [0.870, 0.740, . . . , 0.129].

Multiplying each by the scaling factor 10^3 yields the first elements of the ordered pairs shown in the table.

As explained in the heading of Table 5, the second element in each ordered pair is the value of $[(\rho_r \mathcal{M}_r) \bmod m_e]$.

Since $m_e = 2$ and $\gcd(m_k, m_e) = 1$ for all k, then $\mathcal{M}_r \bmod m_e = 1\ \forall r \Rightarrow (\rho_r \mathcal{M}_r) \bmod 2 = (\rho_r \bmod 2)$, which is a single bit.

In closing we would like to point out the following:

1. We deliberately chose the radix of the computations to be 10 only for the ease of illustration: it is a lot easier to understand how the RAT is created if the entries are decimal numbers (as opposed to binary numbers).

In all the real implementations of the RAT tables (including the RAT tables we created for the GPU experiments) the radix is 2 and the entries in the table are unsigned scaled integers of small word-length (up to approximately 12 bits for a 2048-bit RN system).

2. Note that existing (full-precision) methods would require a huge amount of storage as compared with the RAT above; because each entry of the table would be a long integer with at least 617 decimal digits. Hence, the size of the table (which equivalent to the above RAT) for full precision computations would be approximately $\frac{617}{3} \approx 216$ **times** larger.

References

[1] M. Abtahi, P. Siy, The Factor-2 sign detection algorithm using a core function for RNS, Comput. Math. Appl. 53 (2007) 1455–1463. URL = www.elsiver.com/locate/camwa.

[2] J.-C. Bajard, L.-S. Didier, P. Kornerup, An RNS montgomery modular multiplication algorithm, IEEE Trans. Comput. 46 (1998) 766–776.

[3] J.-C. Bajard, L.-S. Didier, P. Kornerup, Modular multiplication and base extensions in residue number systems, in: In Proceedings of the 15th IEEE Symposium on Computer Arithmetic, IEEE, 2001, pp. 59–65.

[4] J. Bajard, S. Duquesne, M. Ercegovac, N. Meloni, Residue systems efficiency for modular products summation: Application to Elliptic Curves Cryptography, in: Proc. of SPIE, vol. 6313, 2006, pp. 631304–631316.

[5] J. Bajard, M. Kaihara, T. Plantard, Selected RNS bases for modular multiplication, in: Proc. of the 19th IEEE International Symposium on Computer Arithmetic, Portland, Oregon, 2009, pp. 25–32.

[6] D. Banerji, J. Brzozowski, Sign detection in residue number systems, IEEE Trans. Comput. 100 (1969) 313–320.

[7] H. Brönnimann, I. Emiris, V. Pan, S. Pion, Sign determination in residue number systems, Theoret. Comput. Sci. 210 (1999) 173–197.

[8] J. Chiang, M. Lu, A general division algorithm for residue number systems, in: Proc. of the 10th IEEE Symposium on Computer Arithmetic, 1991, pp. 76–83.

[9] R. Conway, J. Nelson, New CRT-based RNS converter using restricted moduli set, IEEE Trans. Comput. 52 (2003) 572–578.

[10] R. Crandall, C. Pomerance, Prime Numbers: A Computational Perspective, Springer, 2005.

[11] G. Dimauro, S. Impedovo, G. Pirlo, A new technique for fast number comparison in residue number system, IEEE Trans. Comput. 42 (1993) 608–612.

[12] Milos D. Ercegovac, Tomas Lang, Digital Arithmetic, Morgan Kaufmann, 2004.

[13] F. Gandino, F. Lamberti, G. Paravati, J. Bajard, P. Montuschi, An algorithmic and architectural study on montgomery exponentiation in RNS, IEEE Trans. Comput. 61 (2012) 1071–1083.

[14] N. Heninger, Z. Durumeric, E. Wustrow, J.A. Halderman, Mining your *P*s and *Q*s: Detection of widespread weak keys in network devices, in: USENIX Security Symposium, 2012, pp. 205–220.

[15] C. Huang, A fully parallel mixed-radix conversion algorithm for residue number applications, IEEE Trans. Comput. C-32 (1983) 398–402.

[16] C. Hung, B. Parhami, An approximate sign detection method for residue numbers and its application to RNS division, in: Computers & Mathematics with Applications, vol. 27, Elsevier, 1994, pp. 23–35.

[17] S. Kawamura, K. Hirano, A fast modular arithmetic algorithm using a residue table, in: Proc. of EUROCRYPT, vol. 88, Springer, 1988, pp. 245–250.

[18] S. Kawamura, M. Koike, F. Sano, A. Shimbo, Cox-rower architecture for fast parallel montgomery multiplication, Lecture Notes in Comput. Sci. (2000) 523–538.

[19] I. Koren, Computer Arithmetic Algorithms, second ed., A K Peters Publishers, Natic, Massachusetts, 2002.

[20] M. Lu, J. Chiang, A novel division algorithm for the residue number system, IEEE Trans. Comput. (1992) 1026–1032.

[21] D. Miller, R. Altschul, J. King, J. Polky, Analysis of the residue class core function of Akushskii, Burcev, and Pak, in: Residue Number System Arithmetic: Modern Applications in Digital Signal Processing, IEEE Press, 1986, pp. 390–401.

[22] Nvidia Pascal Architecture Detailed, last updated in 2016. http://wccftech.com/nvidia-pascal-specs.

[23] B. Parhami, Computer Arithmetic Algorithms and Hardware Designs, Oxford University Press, 2000.

[24] S.J. Piestrak, A high-speed realization of a residue to binary number system converter, IEEE Trans. Circuits Syst. II 42 (1995) 661–663.

[25] K. Posch, R. Posch, Modulo reduction in residue number systems, IEEE Trans. Parallel Distrib. Syst. 6 (1995) 449–454.

[26] Optimal Asymmetric Encryption Padding, last updated in 2015. http://en.wikipedia.org/wiki/optimal_asymmetric_encryption_padding.

[27] A. Shenoy, R. Kumaresan, Fast base extension using a redundant modulus in RNS, IEEE Trans. Comput. (1989) 292–297.

[28] M. Soderstrand, C. Vernia, J.-H. Chang, An improved residue number system digital-to-analog converter, IEEE Trans. Circuits Syst. 30 (1983) 903–907.

[29] L. Sousa, L. Tulisbon, Efficient method for magnitude comparison in rns based on two pairs of conjugate moduli, in: Proc. of the 18th IEEE Symposium on Computer Arithmetic, 2007, ARITH'07, 2007, pp. 240–250.

[30] N. Szabo, R. Tanaka, Residue Arithmetic and its Applications to Computer Technology, McGraw-Hill, 1967.

[31] The "prime-counting function", explained in wikipedia, last updated in 2016. http://en.wikipedia.org/wiki/prime_counting_function.

[32] "The Primorial function" explained in wikipedia, last updated in 2016. http://en.wikipedia.org/wiki/primorial.

[33] T. Van Vu, Efficient implementations of the Chinese remainder theorem for sign detection and residue decoding, IEEE Trans. Comput. 100 (1985) 646–651.

[34] T. Van Vu, The use of residue arithmetic for fault detection in a digital flight control system, NAECON (1984) 634–638.

[35] Y. Wang, X. Song, M. Aboulhamid, A new algorithm for RNS magnitude comparison based on new Chinese remainder theorem II, in: Great Lakes Symposium on VLSI, IEEE Computer Society, 1999, pp. 362–365.

[36] N. Weste, K. Eshraghian, Principles of CMOS VLSI Design, A Systems Perspective, second ed., Addison Wesley, 1993.

[37] D. Younes, P. Steffan, Universal approaches for overflow and sign detection in residue number system based on $\{2^n - 1, 2^n, 2^n + 1\}$, in: ICONS 2013, The Eighth International Conference on Systems, 2013, pp. 77–81.

Dhananjay S. Phatak received his B.Tech. degree from IIT Bombay (Mumbai) in Electrical Engineering; MSEE in Microwave Engineering from Univ of Massachusetts (UMASS) at Amherst; and his Ph.D. in Computer Systems Engineering also from the ECE Dept. at UMASS Amherst in 1994.

He was an Assistant Professor in the ECE Dept. at the State University of New York at Binghamton from Fall 1994 through Spring 2000. Since the fall of 2000, he has been an Associate Professor of Computer Engineering in the CSEE Dept. at UMBC. His research has been supported by NSF, NSA as well as local companies (Aether Systems Inc., Northrup Grumman). He was awarded the NSF CAREER award in 1999. His current Research interests are in Computer Arithmetic Algorithms and their h/w realizations and all aspects of cyber/information/data/computing/network/systems security.

Steven D. Houston received a B.S. in Computer Engineering from Texas A&M University and an M.S. in Computer Science from the University of California, Berkeley.

He is currently a Ph.D. candidate in the CSEE Dept. at University of Maryland, Baltimore County. His research interests span multiple topics, including computer arithmetic, data mining, and security.