PECCA: Pointers with Error Correcting Codes using Arithmetic

Cadin Cross
University of Michigan
Ann Arbor, Michigan, USA
cadinc@umich.edu

Deric Dinu Daniel
University of Michigan
Ann Arbor, Michigan, USA
dericdd@umich.edu

Daniel Werner
University of Michigan
Ann Arbor, Michigan, USA
dbwerner@umich.edu

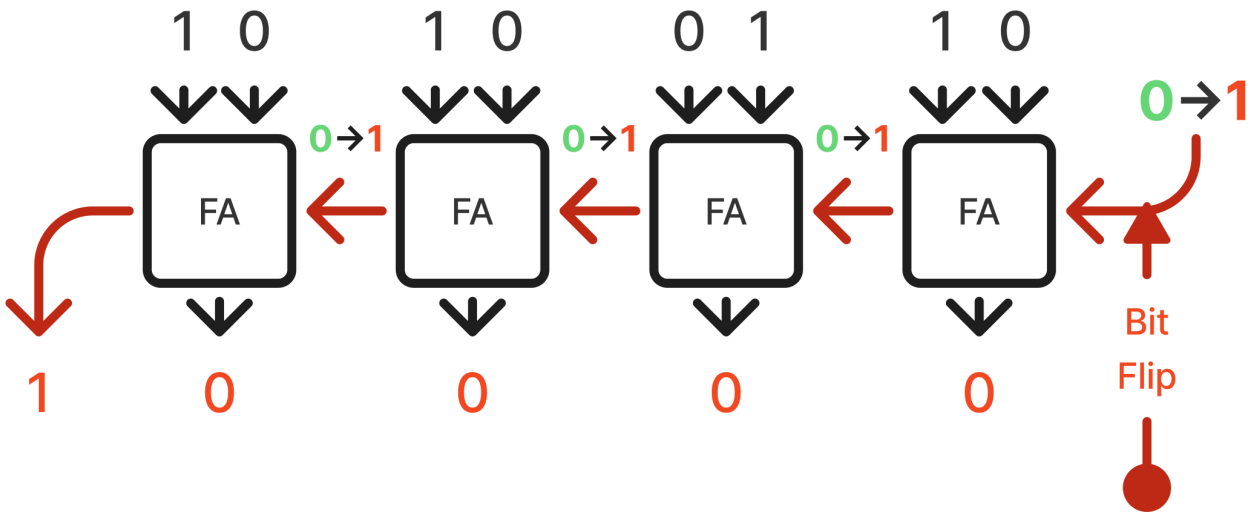

Figure 1: Error Propagation due to a single bit-flip

Abstract

We propose *PECCA*, an ECC scheme utilizing AN Codes to correct single-bit and arithmetic errors caused by transient faults. We analyzed how soft errors can have second order-effects on the results of common arithmetic modules. We also developed a correction and detection algorithm for AN Codes. With our weight generation scheme, we were able to find a PECCA code that could correct up to Arithmetic Distance of 3. Finally, we observed that our correction scheme can correct up to 42-bits of error introduced by a single bit-flip in a multiplier, while still potentially being able to correct more.

Keywords

Reliability, ECC, AN Codes, Hamming Codes, Pointers, Encoding

ACM Reference Format:
Cadin Cross, Deric Dinu Daniel, and Daniel Werner. 2025. PECCA: Pointers with Error Correcting Codes using Arithmetic. In *Proceedings of EECS 573: Microarchitecture (EECS 573 W25)*. ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

EECS 573 W25, Ann Arbor, MI

1 Introduction

One the most important expectations from any computer is reliability and accuracy, for without it, modern computing would not be the industry it is today. An unreliable system leads to unexpected behavior and unforeseen consequences. On the scope of a computer, that can mean creating exceptions caught by software or corrected by internal logic. However, these problems due to unreliability can propagate further into a processor, leading to worse consequences. Outside of the scope of a computer, unreliable behavior can lead to monetary damages, damage to company trust, and even harm for humans interacting directly with the unreliable system.

Often, the very environment in which the system operates decreases the reliability of the system. Transient faults (soft errors) are non-recurring events that impact charge on a transistor and lead to unforeseen behavior, such as a random bit being flipped. Soft errors can be caused by a variety of different sources, such as electromagnetic interference or even neutrons in the atmosphere and in space. For systems under extreme conditions, the chance of transient faults occurring increases, and in turn so does their chance of causing errors to propagate within the system. Additionally, these events can often be silent and go undetected for long periods of operational time, only being detected once they cause unexpected behaviors or system failure. Thus, the need to detect and simultaneously correct these silent data corruptions arises.

Currently, pointers are a weak point terms of transient faults. When a single bit is flipped inside of a pointer's address, it could lead to unforeseen behavior with any instruction that attempts to use that address. As of now, one of the best solutions to checking pointers is Pointer Authentication Codes (PAC's). PAC's are a hardware-based security mechanism designed to protect pointers and control flow in modern processors. PAC's work by cryptographically authenticating pointers using a secret key and a cryptographic function to generate a signature. This signature is embedded within the unused high-order bits of the pointer, allowing processors to verify the integrity and authenticity of pointers during runtime. If a pointer is modified by an attacker, the integrity check will detect the fault and cause an exception, ending the program. This solution has been implemented in Armv8.3-A and above architectures (including Apple Silicon devices). PAC's have been proven to be effective for thwarting attacks; however, there is no ability for the program in fault to recover.

2 Related and Previous Works

Reliability is a well-studied topic in the field of Computer Architecture. Several disciplines and applications, such as space, medical, automotive, and elections, necessitate that these systems operate correctly. We note a survey [7] and overview on the currently used methodologies for implementing Error Correction and Detection in Embedded Systems.

Previous work has also been made for identifying vulnerable parts of a processor and protecting them against transient faults [4]. Additionally this work extends to identifying (in a mostly non-intrusive manner) how these errors propagate throughout a system. [5] describes a methodology for observing how errors propagate while under directed radiation testing. We note that there is currently no work done in identifying the propagation of errors at RTL or without using a physical design; as such, we had to implement our own solution to simulate the expected gate-level behavior.

A common error correction and detection scheme is the Hamming Code [1]. Hamming Codes are commonly used to encode and protect data under noisy channels or with strict reliability concerns. We will offer further discussion into how the Hamming Code works and why it does not fit our use case of encoding pointers for reliability.

Our paper focuses on AN Codes, an encoding scheme which we will describe in greater detail later. [3] delves into theory behind AN Codes and how they are used to ensure reliability in arithmetic units. Some work has been done on applying AN Codes in software and hardware beyond their introduction [6]. We note that our application of AN Codes is specific to pointers and offers a more comprehensive analysis on arithmetic modules generating arithmetic errors. We also considered [2], which does offer analysis on AN Codes and an alternate version of them. However, we note that this paper dismisses any analysis on arithmetic errors and only considers single bit-flips, which is inherently prohibitive on expounding the true benefits of AN Codes.

3 Error Correcting Codes

Error Correcting Codes (ECC) are a way to detect and correct Soft Errors - commonly targeting single bit-flips - that might arise in a highly volatile environment. All ECC schemes work by taking the original data (word) and then applying a code on top of it to transform it into an encoded form. The encoded form of the word is known as a codeword. Using this codeword, the ECC scheme can detect if the data has been corrupted by an unexpected event by checking if the codeword exists in the codebook. We will describe at a high level how ECC's detect and correct these errors. Within this paper we discuss ECC schemes and the codes they operate using their binary notation.

3.1 ECC Background

We define the *codebook* of a code as the set of codewords that can be generated using a given code on a range. For example, consider a code M and a finite set of words in the range $[A, B]$. A code, according to its ECC scheme, applies a transformation $M(A)$ to each word, where $M(N)$ is the encoding operation for a word N. The set of all words from A to B with the encoding operation applied (i.e., $M(A), M(A+1), ..., M(B)$) becomes the codebook of M.

ECC schemes perform their correction and detection methods by first evaluating if a given word is within their codebook. Given a codebook S and a word A, if $A \in S$, then A is a valid codeword and requires no correction or detection. If $A \notin S$, then A is not a valid codeword and must be accordingly for the ECC scheme that created the codebook. When $A \notin S$, A is defined as an error code.

However, depending on the ECC scheme and the code chosen, an error code might appear as a valid codeword. Consider what would occur if there existed two codewords in a codebook: 00 and 11. If a user generated the codeword 00 and a random bit flip occurred - transforming the codeword to 01 - the ECC scheme would detect the error code and perform its correction or detection procedures. However, if a subsequent bit-flip occurred - further transforming the codeword to 11 - the ECC scheme would be unable to correct the error as it would see the codeword as belonging to its codebook. Even though the value generated by the user is now incorrect and could potentially cause an error, the ECC scheme would be unable to correct or detect the error. Thus, choosing a proper ECC scheme and code is vital to ensuring correctness.

3.2 Overview of Hamming Codes

A commonly used ECC scheme is Hamming encoding. Hamming codes are used in a range of applications, such as ensuring correctness in a noisy data channel or securing DRAM against soft errors. Hamming codes work by applying the encoding scheme on top of the original data. Instead of directly transforming the original code, the encoding is a series of bits that is concatenated onto the original code.

The Hamming (7,4) code is a widely used form of the Hamming code scheme. It applies 3 parity bits onto 4 bits of data to create a codeword that is 7 bits long. As our paper does not focus on Hamming Codes, we will not go in depth as to how the Hamming Code correction and detection scheme functions. For our work, the most important part of the scheme is the Single-Error Correcting and Double-Error Detecting (SECDED) properties of the Hamming Code. If a bit-flip occurs in a Hamming encoded value, it can be corrected using a parity check. If two bit flips occur, the error cannot

be corrected but may be detected and used to trigger an exception for the OS to handle.

3.3 Hamming Weight and Distance

Hamming Weight - for the purposes of the binary representations of all codewords in this paper - is defined as the number of "1" bits in a codeword. For example, the decimal number "12" has a Hamming Weight of 2, because the binary representation of 12 (1100) has two non-zero bits.

Hamming Distance is defined in relation to two codewords. It is the absolute value of the difference in their Hamming Weights. For example, the Hamming Distance d between codewords 0010 and 1011 is $|1-3|$, which means $d = 2$.

The distance d of a codeword can be used to find up to many errors a given code can correct up. The maximum correctable distance D a given code can correct up is defined as

$$D = \lfloor \frac{d-1}{2} \rfloor$$

For example, the distance d between all elements of the codebook of Hamming (7,4) codes is 3. Given the equation above, Hamming (7,4) codes can correct up to only one error. For Hamming codes, the error is a bit-flip, or an error that generates a Hamming Distance of 1 between the original codeword and the error code.

3.4 Pitfalls of Hamming Codes for Use with Pointers

SECDED codes, such as the Hamming (7,4) code, cannot correct or detect an error code that arose from an error or bit-flip of 3 or more bits. This is because the error will result in another word that is in the codebook of the Hamming (7,4) code, as outlined earlier.

We observe that because of the inherent properties of Hamming codes, they cannot be easily applied to secure pointers against transient faults. Pointers are also often the subject of arithmetic operations (addition, subtraction, multiplication, etc.) that must be applied to access the correct memory addresses which the user requires. If a pointer were to be encoded using a Hamming code, performing these operations would break the encoding of the parity bits. Furthermore, encoding the values that will be used in the operation will not ensure that the encoding will still be valid for the end result. Hamming codes are a form of linear block code, but are not in the same linear space as the arithmetic operations applied to pointers.

Thus, the arithmetic operations that pointers are subject to necessitate an encoding scheme which operates on the same linear space as the aforementioned operations.

4 Arithmetic Errors

In this paper, we propose the usage of AN Codes, an ECC scheme that is compatible with pointer arithmetic. AN Codes function on the same linear space as the operations applied to pointers, so long as the operands are also encoded. In order to additionally motivate the usage of AN-codes, we will discuss further pitfalls of traditionally used Hamming codes by observing a second-order effect of single-bit flips.

4.1 Error Ripple Model

The effects of transient faults are not limited solely by the scope of registers or memory. We argue that logic elements in a processor are significantly more vulnerable to memory elements. We illustrate this by offering the example of unexpected behavior within a multiplexor induced by a bit-flip.

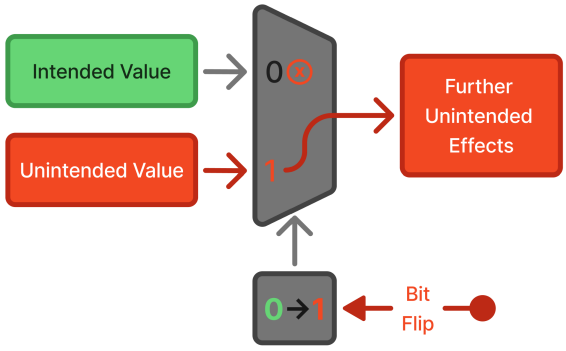

Figure 2: Error Propagation due to a Single-Bit flip

Consider what would occur if a transient fault were to flip the select bit on a multiplexer. This could cause an unintended value to propagate further into the program, leading to further unintended effects. Although later logic has a chance to mask out the unintended propagated value, there is also a chance that the error might still continue further into other dependent logical or memory elements. As a consequence, this invalid result may be latched into a memory element.

We also note that on logic elements, such as a multiplexer, there is no inherent Error-Correcting capability present. A memory element may be protected by ECC, but logic elements are unprotected and, as such, are more vulnerable to transient faults.

We introduce the Error Ripple Model by analyzing the effects of a bit-flip on the carry bits of a Ripple-Carry Adder.

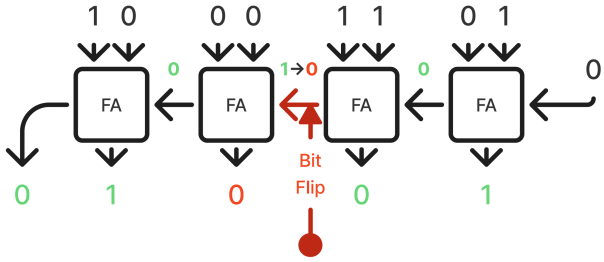

Figure 3: Error Ripple due to a Bit-flip in a Ripple-Carry Adder

In Figure 3, we illustrate what would occur if a bit-flip caused a gate in the carry generation to output the incorrect value of carry out. This example shows an arithmetic error of 1; that is, a

single error in the arithmetic has occurred and has resulted in a Hamming Distance of 1 between the expected and actual result of the operation.

When a single-bit flip occurs in a logical gate, the Hamming Distance between the expected and actual result of the operation may be much greater than one. We will discuss our analysis and findings later about how a single error can ripple throughout an arithmetic module and the way in which Hamming Codes cannot recover from such errors.

4.2 Arithmetic Weight and Distance

Before our analysis of how arithmetic errors can propagate in commonly used modules, we must first discuss Arithmetic Weight and Distance, as they are a vital part of this analysis and our work with AN codes.

Arithmetic Weight is defined as the minimum number of terms required to represent a number in standard polynomial form. Since we use binary representations of codewords in this paper, we will discuss this in terms of binary. The Arithmetic Weight is the number of non-zero bits in that minimum representation.

To determine the arithmetic weight of a number, one must start with its decimal representation. For example, take the codeword that is represented by the decimal number 7. In binary form, it can be represented as 0111, since $7 = 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$. The number of non-zero terms in this polynomial representation of 7 is 3. However, 7 may be represented in another form. It can be written as 1001, since $7 = 1 * 2^3 - 0 * 2^2 - 0 * 2^1 - 1 * 2^0$, the number of nonzero terms in this polynomial representation is 2. The minimum number of terms that can be used to represent 7 is 2; therefore, the Arithmetic Weight w of 7 is 2.

Arithmetic Distance is defined as the weight of the difference between two numbers. To find the Arithmetic Distance between two codewords x and y, simply find their difference $x - y$, and then take the Arithmetic Weight w of their difference: $w(x - y)$.

4.3 Arithmetic Error Testing Methodology

We analyzed how a single bit-flip would cause errors to ripple throughout a common arithmetic component. In order to do this, we created a custom simulation in C++ that would mimic the logic-level behavior of gates and wires. We did not simulate any transistor-level components of the gate, as we were only concerned with the outputs of a bit-flip affecting further gates. We also do not consider metastability within our simulation. Our simulation allowed us to create distinct gates and propagate values from gate-to-gate. Additionally, it allowed us to insert bit-flips on a specific gate to observe the effects of how a bit-flip would propagate to the final result.

To gain some insight into how a single-bit flip could impact the result of a common module, we tested how errors would propagate on three different arithmetic modules: a 64-bit Ripple-Carry Adder, a 64-bit Kogge-Stone Adder, and a 64-bit (32-bit inputs) Carry-Save Multiplier. Our reasoning was that these arithmetic operations would be commonly applied on pointers and as such, we wanted to observe how Error Rippling would affect the final result of these pointer operations.

Our testing method consisted of multiple repetitions of iterating through every gate in the module and flipping the result of the logical operation of a gate. We utilized random inputs within the range of the possible inputs for the module. We would then observe the Hamming Distance and the Arithmetic Distance between the observed result and the expected result.

4.4 Error Propagation of the Ripple-Carry Adder

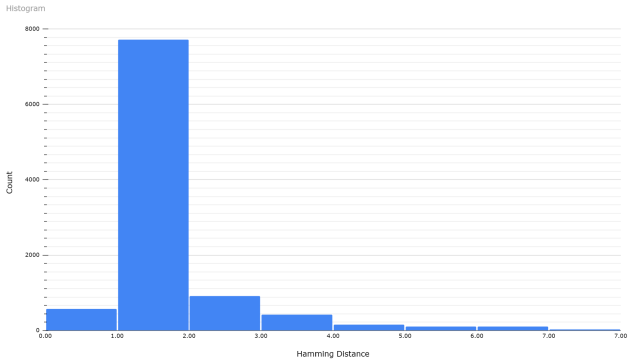

Figure 4: Histogram of Error Hamming Distances for RCA

In Figure 4 we plot the histogram of Hamming Distance between the actual and expected result. Out of 10,011 total bit-flips, 17.14% of the gate bit-flips in the RCA resulted in a Hamming Distance greater than 1, meaning these errors could not be recovered by normal Hamming Code means.

We also analyzed the Arithmetic Distance between the actual and expected result. For all gate bit-flips, the only two observed Arithmetic Distance values were 0 and 1. 5.71% of bit-flips resulted in Arithmetic Distance of 0, and 94.29% of bit-flips resulted in Arithmetic Distance of 1.

4.5 Error Propagation of the Kogge-Stone Adder

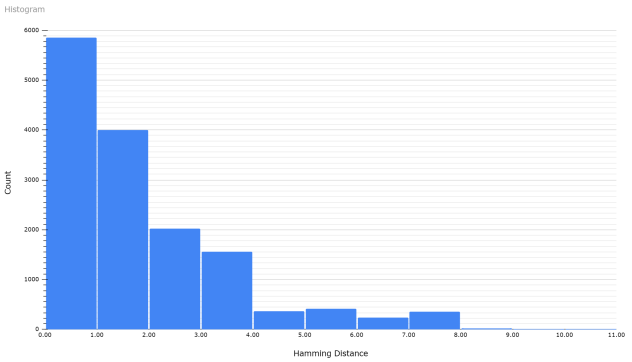

Figure 5: Histogram of Error Hamming Distances for KSA

We plot the histogram of Hamming Distances in Figure 5 for the Kogge-Stone Adder. In contrast to the RCA, the KSA's bit-flip

Table 1: Gate Count of the Arithmetic Modules

Module	AND Gates	OR Gates	XOR Gates
Ripple-Carry Adder	128	64	193
Kogge-Stone Adder	770	385	192
Carry-Save Multiplier	3072	1024	2048

injection results in a Hamming Distance of 0 in 39.50% of the 14,818 cases. We believe that this large amount of low Hamming Distances might be caused by the Group Propagate logic masking out bit-flips.

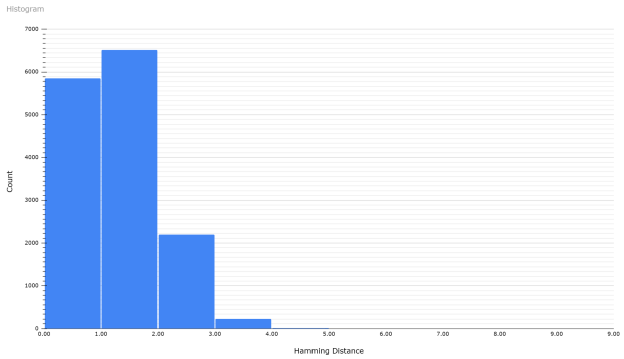

Figure 6: Histogram of Error Arithmetic Distances for KSA

Additionally, in Figure 6, we plot the histogram of the Arithmetic Distance between the Actual and Expected Result. Unlike the RCA, the Arithmetic Distances are not exclusive to the values 0 and 1. We observe that 16.55% of gate bit-flips result in an Arithmetic Distance greater than 1. The maximum Arithmetic Distance observed in our testing was 9.

This increased Arithmetic Distance could imply that the architecture of the Kogge-Stone Adder as compared to the Ripple-Carry Adder is what leads to an increased Arithmetic Error. Although the KSA tends to mask faults more frequently than the RCA, the faults which manage to propagate through have more pronounced effects on the final result due to the Group Generate and Propagate structures.

4.6 Error Propagation of the Carry-Save Multiplier

In Figure 7, we have also plotted the histogram of Hamming Distance between the actual and expected result. Out of 12289 total bit-flips, 51.64% resulted in a Hamming Distance greater than 1. 28.74% of cases resulted in a Hamming Distance greater than 2, meaning these errors would go undetected by the Hamming (7,4) Code scheme.

For the Arithmetic Distance between the actual and the expected result, it also only resulted in Arithmetic Distance values of 0 and 1. We believe that this is due to the fact that the Carry-Save Multiplier is built using Carry-Save Adders, which are built using Ripple-Carry Adders. As such, the similar internal linear architecture leads to a maximum of Arithmetic Error of 1. 3.91% of bit-flips resulted in

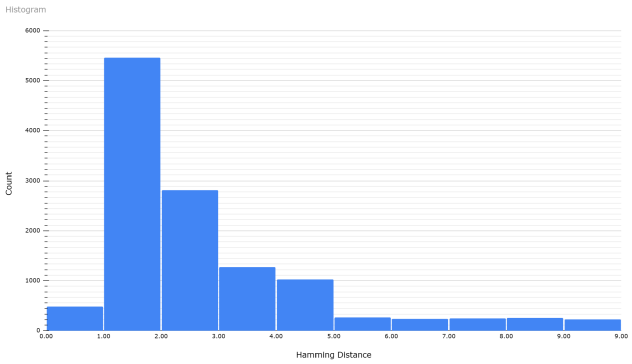

Figure 7: Histogram of Error Hamming Distances for CSM

an Arithmetic Distance of 0, and 96.09% of bit-flips resulted in an Arithmetic Distance of 1.

5 AN Codes

In this paper, we argue for the usage of AN Codes for use in Error Correcting methodologies for pointers. AN Codes operate on the same linear space as the operations that are applied to pointers, meaning that the encoding of the pointer will be preserved so long as the operand is encoded with the same AN Code. Additionally, a code using the AN Code scheme can correct more than a single bit-flip as well as recover from multi-bit errors that arise from arithmetic errors, as previously shown in the Error Ripple Model.

5.1 Overview of AN Codes

AN Codes are an encoding scheme that uses a number, A, to encode a word. To ensure clarity, we write all A encoding values in decimal form, except when writing the encoding or decoding procedure.

The encoding operation is as follows: to encode a word, multiply it by A. For example, if the code 13 is used, and the word that must be encoded is 00001010, then the codeword is: $1101 * 00001010 = 10000010$.

The decoding operation is as follows: to decode a codeword, divide it by A. For example, if the code is 13, and the codeword is 001001111101, then the word is $001001111101/1101 = 00110001$.

To demonstrate how AN codes are suited for use with pointers, we illustrate by example as to how AN codes can be used in conjunction with arithmetic operations.

Consider the code 7, which was used to generate the codeword $ptr = 001001000101$ from the word 01010011. If the operation $ptr + 0100$ is meant to be applied to it, all of the operands must be encoded using the same AN Code. This means that the operand 0100 must be converted to 00011100 by means of multiplying the original operand by 7. At this point, the operation can be applied since all operands are encoded by the AN Code. The result will then be $001001000101 + 00011100 = 001001100001$. Thus, $ptr = 001001100001$. If we decode the value using the AN Code, the result is 01010111, which is the expected result from adding together the non-encoded values.

5.2 Range of AN Code Values

The range of possible values for a code A must be carefully considered before analysis. The maximum value for the code A is 2^{m-n}, where m is the maximum number of bits that a codeword can use, and n is the maximum number of bits used by the words which will populate the codebook.

For the purposes of our paper, we use 48-bit pointers that are used in a 64-bit space. Therefore, our range of AN Code values is $[1, 2^{16}]$. However, using the code 1 is the same as applying no encoding due to the way the encoding and decoding methods work.

5.3 Error Correction and Detection Algorithm

The AN Code correction and detection algorithm is more complex than the Hamming Code algorithm of checking parity bits. We now discuss our implementation of the correction algorithm for AN Codes.

Similar to other ECC schemes, the maximum correctable distance D of an AN code is given as $\lfloor \frac{d-1}{2} \rfloor$. For AN Codes, the distance d is the minimal Arithmetic Distance, not Hamming Distance.

To detect an arithmetic error for an A value of A, we can check if $codeword \,\%\, A == 0$. If the modulo result is not 0, then the codeword is not in the codebook for A and an error must have occurred.

In order to correct an arithmetic error, we must find the codeword with the smallest Arithmetic Distance from the error code. This is similar to how other ECC schemes such as Hamming (7,4) Codes correct their errors. However, the algorithm for finding the nearest codeword is more complex for AN Codes.

We use the following formula to find the codeword with the nearest Arithmetic Distance:

Algorithm 1 AN Code Correction

```
1:  function CorrectPtr(enc_ptr, enc, D, arithWeightNums:
    map<arith. weight, array<int»)
2:      if enc_ptr mod enc = 0 then
3:          return enc_ptr
4:      end if
5:      for i ← 0 to D − 1 do
6:          for j ← 0 to |arithWeightNums[i]| − 1 do
7:              check_add ← enc_ptr + arithWeightNums[i][j]
8:              check_sub ← enc_ptr − arithWeightNums[i][j]
9:              if check_add mod enc = 0 then
10:                 return check_add
11:             end if
12:             if check_sub mod enc = 0 then
13:                 return check_sub
14:             end if
15:         end for
16:     end for
17:     return −1                                   ▷ Error
18: end function
```

To find the nearest arithmetic codeword, we must take our error code and both add and subtract a number with a given arithmetic weight to the error code. We want to find the arithmetically closest codeword so we start with numbers with Arithmetic Weight 1 and go up until the maximum arithmetic weight D that A (also referred to as *enc*) is allowed to correct.

Table 2: Count of numbers in Arithmetic Weight Bins

Arithmetic Weight	Count
1	126
2	5859
3	120837
4	1825859
5	21679273
6	210855079

5.4 Methodology for Generating Weights

To know the Arithmetic Weights that a given AN Code could add or subtract to attempt correction, we first had to generate *arithWeightNums*, a map that contains numbers sorted into bins of Arithmetic Weights. To create the items that would populate a bin of Arithmetic Weight w, we would generate all combinations of strings given a certain Hamming Weight w. This was then used to calculate all numbers whose standard polynomial representation match that string. If the calculated number already existed in a lower bin we discarded the value, otherwise we inserted it into the current bin.

We utilized this algorithm to generate 64-bit numbers, as our pointers would be encoded using up to a 64 bit codeword even though they only took up 48 bits.

In Table 2, we list the number of words in the first 6 Arithmetic Weight bins we generated. We emphasize the increasing amount of numbers per each bin, and how the increasing bin sizes might limit the speed of the correction algorithm as currently implemented.

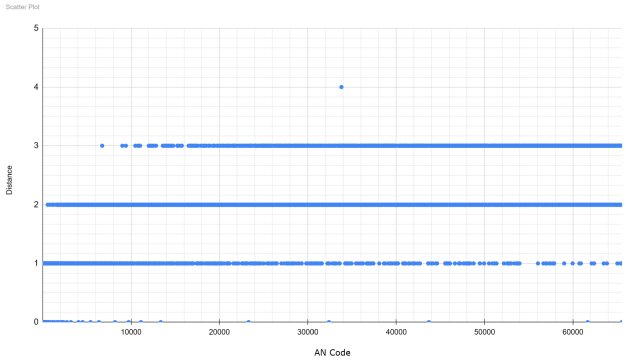

Figure 8: D for all primes in the range $[2, 2^{16}]$

5.5 Maximum Correctable Distance

We now describe the methodology for finding the maximum correctable distance D of an AN Code. Given an AN Code A, iterate through all the values C in a bin of Arithmetic Weight w. If, for any value in the bin Arithmetic Weight w, $C \,\%\, A == 0$, then the maximum correctable distance of A is $w - 1$. For example, if you are

Table 3: AN Codes with Maximum Correctable Distance *D*

Max. Correctable Dist.	Count
0	67
1	872
2	4032
3	1571
4	1

iterating through the elements of bin 3, and you find an element which A is a factor of, then D is 2.

Using the algorithm described above, we plot the maximum correctable distance D for all prime numbers in the range $[2, 2^{16}]$. in Figure 8. To further expound on this data, we use Table 3 to show the number of AN Codes with had a given maximum correctable distance D.

There were no prime numbers in our range which had a D larger than 4 given our 64 bit space. The prime number with the highest D of 4 was 33791. As such, generating the bins of Arithmetic Weights larger than 5 is unnecessary, and using the bins of Arithmetic Weights larger than 4 in the correction algorithm is also not needed as no AN Code will correct using the bins beyond 4.

6 AN Code Analysis

In a full-system implementation of AN encoded pointers, it is recommended that only a single AN code be chosen to encode all values. This is because performing operations using values that are encoded with different AN values will lead to malformed results. Additionally, the overhead of keeping track of which pointer is encoded with which respective AN Code would be unwieldy and unmanageable.

Furthermore, we imagine a constant AN code could be optimized within the hardware to reduce the overhead of the encoding scheme. This could lead to a negligible CPI impact on the encoding and decoding of values. However, confirmation of this belief is left for later work.

Therefore, we first conducted our research with a search for the best AN Code values for protecting against single bit-flips. We did this to establish the baseline for comparison with the SECDED behavior of Hamming (7,4) Codes. Next, we took to our C++ Gate-Level Simulation to observe how our AN Codes could be used to correct errors in our previously analyzed arithmetic modules.

6.1 Single-Bit Error Analysis

We will briefly discuss our initial search to see how AN Codes could protect against single bit-flips. To do this, we created a templated C++ class that behaved syntactically like a pointer. This class would automatically manage the encoding and decoding of words as needed, and allowed for single bit error injection. Our stress testing methodology consisted of injecting a single bit error every time a pointer access was made, and then flipping a random bit in the 64-bit codeword. We tested this on a modified benchmark suite from our EECS 470 class that increased the number of total pointer accesses.

Our method for correction in the single-bit error analysis was to first check the modulo of the codeword with the AN code. If it was not zero, then a bit-flip had occurred and it required us to step through every bit of the codeword, test flipping every bit until the modulo of the codeword was 0.

We observed that naively searching for AN values by testing all codes would not yield an objective best AN code or observable pattern. However, we did observe some takeaways. Encoding a word with a power of 2 led to a 0% success rate since a bit-flip would eventually occur on the exact bit that made the modulo of the code 0.

Searching through all prime numbers in the range $[2, 2^{16}]$ yielded more comprehensive results.

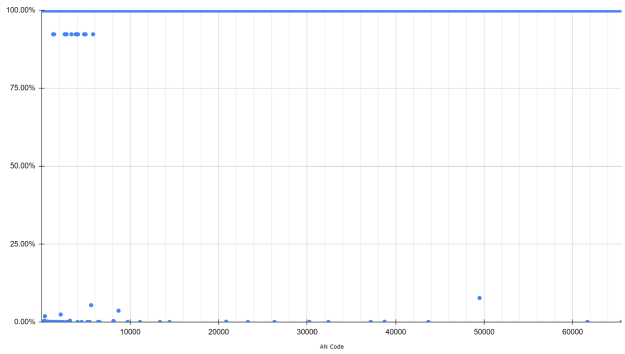

Figure 9: Success % of all Primes in the Range $[2, 2^{16}]$

Out of the 6542 prime numbers in our range, 6420 (98.12%) had a 100% Success Rate on Single-Bit flip errors. The larger the prime number, the more likely its codebook will consist of unique and distant codewords.

We make the note here that from this point of the paper until the end, all of our AN Code analysis used the set of all prime numbers in the range $[2, 2^{16}]$. Searching through a set that was proven to have a vast majority of numbers that were on par with SECDED schemes led to more productive and insightful results in our proceeding error analysis.

6.2 Ripple Carry Adder AN Code Analysis

We now present our AN Code analysis on our Ripple Carry Adder. As observed previously, the maximum arithmetic distance of the generated result and expected result of the RCA is 1, no matter the AN Code used to encode the values.

As shown in Figure 10, the average Arithmetic Distance is lower than the average Hamming Distance for all AN Codes. The average Hamming Distance is greater than 1 in all cases of AN codes, meaning that a SECDED scheme such as Hamming (7,4) Codes could not recover from all bit-flip errors introduced in a RCA.

In our analysis shown in Figure 11, we plot the rates for relevant data in RCA. We iterated through every gate on the RCA, flipping the result of the gate and observing whether the AN Code would be able to correct the value. We observed the following: relevant rates for the bit-flips that caused no change to the Hamming Distance or Arithmetic Distance; the bit-flips errors that were recovered

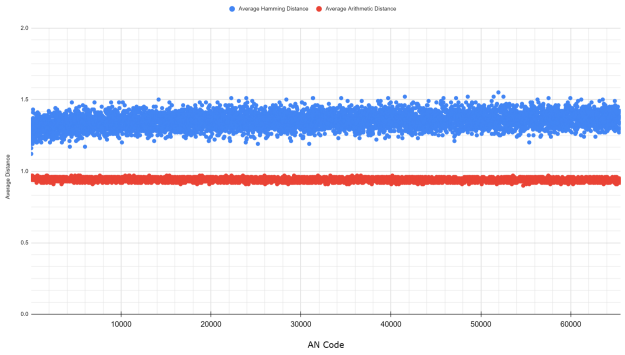

Figure 10: Avg. Hamming and Arithmetic Distances for RCA

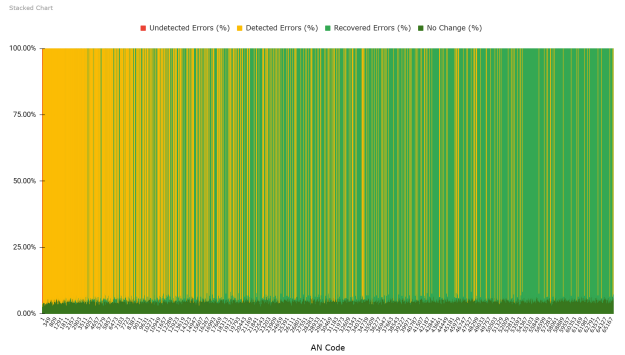

Figure 11: Recovery Rates for AN Codes on the RCA

successfully by the AN Code; the bit-flip errors that were detected by the AN Code and caused an exception; and the bit-flips that were undetected by the AN Code and would have caused an error later on.

An AN Code that can fix and detect more errors than leave them undetected is a good choice to use in an encoding scheme. Ideally, a perfect AN Code would be able to recover from all errors introduced.

In our results, we observe that most large AN Codes can either correct 100% of the errors introduced or detect 100% of the errors. In the RCA, this seems to be mutually exclusive. As expected, larger AN Code values are able to more reliably correct all errors than smaller AN Code values.

6.3 Kogge-Stone Adder AN Code Analysis

For the Kogge-Stone Adder, the average of the maximum arithmetic distance observed per AN Code was 22.03. This somewhat precludes our analysis on the recovery behavior of AN Codes, as no AN Code will be able to correct an error of such a high arithmetic distance.

Figure 12 plots the Average Hamming Distance and Arithmetic Distance between the actual and expected result of the KSA. For the first prime numbers from $2 \rightarrow 1481$, there is a distinct increase in both the Average Hamming and Arithmetic Distances. However,

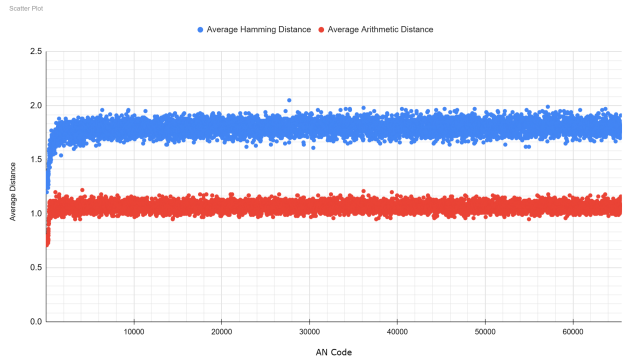

Figure 12: Avg. Hamming and Arithmetic Distances for KSA

this does not correlate with a higher chance of success than all larger AN prime numbers.

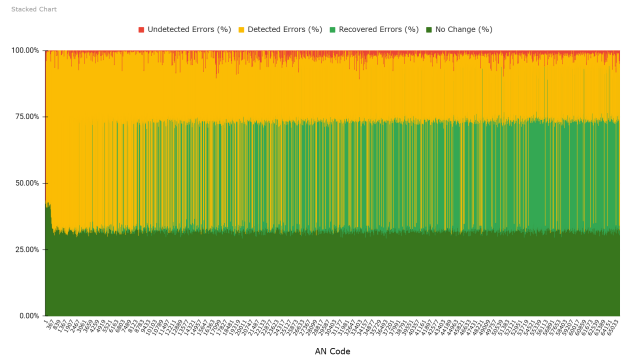

Figure 13: Recovery Rates for AN Codes on the KSA

The same error recovery rates as plotted for the RCA are plotted for the KSA in Figure 13. As expected with the analysis for the Error Propagation of the KSA, around 30% of the gate errors introduced required no recovery. We also again observe that larger AN values tend to be more likely to recover or detect from more errors.

According to our standards, a good AN Code would be one that had 0 undetectable errors, minimized the number of detected errors, and recovered the majority of errors.

There was not a single AN-Code in our range of prime numbers from $[2, 2^{16}]$ was 100% successful (no undetected or detected errors). The single highest Success % was for the AN Code value of 44543, which recovered 63.03% of errors, and detected 3.27%, suffered 0.45% undetected errors, and the remaining 33.26% were bit-flips that had no effect on the final result.

The AN Code that had no undetectable errors and had the highest Success % out of this standard was 64451. This AN Code had a recovery rate of 62.14%, detected but failed recovery error rate of 5.12%, an undetected error rate of 0%, and a no effect rate of 32.74%.

We again note that the high rate of uncorrectable errors in the Kogge-Stone Adder are because of its architectural differences from the Ripple-Carry Adder. The linear carry structure of the RCA leads to a predictable error that is in line with the model used to delineate

the alternative representation of a number used for its Arithmetic Weight. Because of the tree-based structure of the carry-out of the KSA, the errors do not fall in line with the previous model of expected errors, and as such lead to increased Arithmetic Distances.

6.4 Carry-Save Multiplier AN Code Analysis

For the Carry-Save Multiplier, we were not able to plot the entire previously used range of AN Code values due to time constraints and a temporally-inefficient testing methodology. However, the results generated are conclusive enough to show the pattern present in the Carry-Save Multiplier.

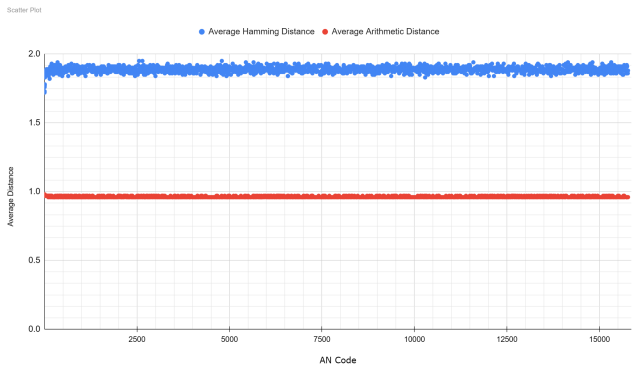

Figure 14: Avg. Hamming and Arithmetic Distances for CSM

We plot the average Hamming Distances and average Arithmetic Distance for the Carry-Save Multiplier in Figure 14. Similarly to the other arithmetic modules observed, the average Hamming Distance is always larger than the average Arithmetic Distance.

The only observed values for Arithmetic Distance in the Carry-Save Multiplier were 0 and 1. This is due to the fact that the Carry-Save Multiplier has a linear carry-out structure similar to the Ripple-Carry Adder.

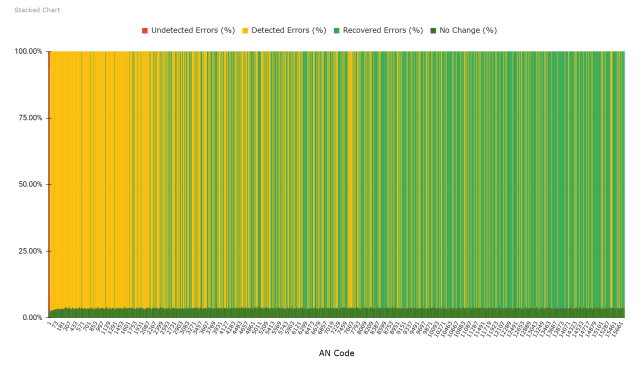

Figure 15: Recovery Rates for AN Codes on the CSM

Our Error Recovery rates for each respective AN Code using the Carry-Save Multiplier is plotted in Figure 15. Similar to the RCA, there is a low chance (< 5%) that the error will be masked out by proceeding logic. Additionally, we observe that most AN Codes can either correct 100% of the errors introduced or detect 100% of errors.

Using a SECDED scheme such as Hamming (7,4) Codes would leave most errors in the Carry-Save Multiplier undetectable and uncorrectable, allowing them to cause further unintended effects when latched into memory elements.

For the Carry-Save Multiplier, the maximum Hamming Distance that we observed our AN Code scheme correct was 42. This was done with a relatively low AN Code value, 7177. We note that this is not part of our analysis of AN Codes with respect to one another; rather it emphasizes the correction performance of AN Codes over their Hamming (7,4) counterpart.

7 Conclusion

Our work introduced AN Codes specifically for the purpose of encoding 48-bit pointers in a 64-bit space. To motivate the usage of AN Codes over a common ECC scheme such as Hamming Codes, we outlined how common Hamming Codes were incompatible with pointer arithmetic and how the Error Ripple Model shows how transient faults may have more pronounced effects than a single bit-flip in a memory element.

We outlined the practical usage of both Arithmetic and Hamming distances and weights and created a C++ gate-level simulation to identify and analyze error propagation in three common arithmetic modules.

Our work also developed an error correction and detection algorithm for AN-Codes, something not commonly discussed in the existing literature. We then used our algorithm to analyze how our set of primes in the range $[2, 2^{16}]$ performed with correcting and detecting errors beyond the limits of a Hamming Code scheme. We highlight that we were able to observe a 42-bit correction on an error in the multiplier.

8 Future Work

We propose extending PECCA to work across the full stack: Software → Operating System → Compiler → Hardware. On the software side of the stack, this means extending our C++ templated class (used in the single-bit error analysis) to have specific compiler support for explicit programmer usage of encoding and protecting values.

Further analysis could be undertaken in other common logical and arithmetic modules to see how errors affect them - and how AN Codes fare with correcting them. Our initial analysis did not include the KSA, but after evaluation, it gave us more insight into how the structure of combinational logic can lead to increased errors. An interesting consequence of this is that trade-offs between speed and reliability are made even at the most basic architectural levels.

Regarding speed, we would like to see if it is possible to develop a faster way to generate the numbers for each arithmetic weight as well as the actual correction algorithm itself. Currently, correction involves brute force searching for the nearest codeword, which is quite slow. Future work would hopefully find a method to reduce the search space or even a completely different algorithm that does not involve brute force.

Acknowledgments

To Nathaniel Bleier, for his knowledge and insight through multiple classes and for saying "free rizzy table".
To Bradley Schulz and Mustafa Miyaziwala for providing guidance above and beyond expectations.

References

[1] R. W. Hamming. 1950. Error detecting and error correcting codes. *The Bell System Technical Journal* 29, 2 (1950), 147–160. doi:10.1002/j.1538-7305.1950.tb00463.x

[2] Martin Hoffmann, Peter Ulbrich, Christian J. Dietrich, Horst Schirmeier, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2014. Experiences with software-based soft-error mitigation using AN codes. *Software Quality Journal* 24 (2014), 87 – 113. https://api.semanticscholar.org/CorpusID:16971649

[3] James L. Massey and Oscar N. García. 1972. *Error-Correcting Codes in Computer Arithmetic.* Springer US, Boston, MA, 273–326. doi:10.1007/978-1-4615-9053-8_5

[4] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 29–40. doi:10.1109/MICRO.2003.1253181

[5] Manuel Peña-Fernández, Borja Verdasco, Luis Entrena, and Almudena Lindoso. 2025. Soft-Error Detection and Execution Observation for ARM Microprocessors. *IEEE Transactions on Nuclear Science* 72, 4 (2025), 1504–1512. doi:10.1109/TNS.2025.3550369

[6] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. 2010. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *Computer Safety, Reliability, and Security*, Erwin Schoitsch (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–182.

[7] Venu Babu Thati, Jens Vankeirsbilck, Jeroen Boydens, and Davy Pissoort. 2017. Data Error Detection and Recovery in Embedded Systems : a Literature Review. *Advances in Science, Technology and Engineering Systems Journal* 2, 3 (2017), 623–633. http://astesj.com/v02/i03/p80/