AHEAD: Adaptable Data Hardening for On-the-Fly Hardware Error Detection during Database Query Processing

Till Kolditz, Dirk Habich, Wolfgang Lehner
Database Systems Group
Technische Universität Dresden
[first.last]@tu-dresden.de

Matthias Werner
Center for Information Services and High Performance Comp.
Technische Universität Dresden
matthias.werner1@tu-dresden.de

S.T.J. de Bruijn
NubiloSoft
The Netherlands
stefan@nubilosoft.com

ABSTRACT

We have already known for a long time that hardware components are not perfect and soft errors in terms of single bit flips happen all the time. Up to now, these single bit flips are mainly addressed in hardware using general-purpose protection techniques. However, recent studies have shown that all future hardware components become less and less reliable in total and multi-bit flips are occurring regularly rather than exceptionally. Additionally, hardware aging effects will lead to error models that change during run-time. Scaling hardware-based protection techniques to cover changing multi-bit flips is possible, but this introduces large performance, chip area, and power overheads, which will become non-affordable in the future. To tackle that, an emerging research direction is employing protection techniques in higher software layers like compilers or applications. The available knowledge at these layers can be efficiently used to specialize and adapt protection techniques. Thus, we propose a novel adaptable and on-the-fly hardware error detection approach called *AHEAD* for database systems in this paper. *AHEAD* provides configurable error detection in an end-to-end fashion and reduces the overhead (storage and computation) compared to other techniques at this level. Our approach uses an arithmetic error coding technique which allows query processing to completely work on hardened data on the one hand. On the other hand, this enables on-the-fly detection during query processing of (i) errors that modify data stored in memory or transferred on an interconnect and (ii) errors induced during computations. Our exhaustive evaluation clearly shows the benefits of our *AHEAD* approach.

CCS CONCEPTS

• **General and reference → Reliability**; • **Information systems → Record and buffer management**; **Main memory engines**;

KEYWORDS

Reliability, Error Detection, Database Systems, Query Processing

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183740

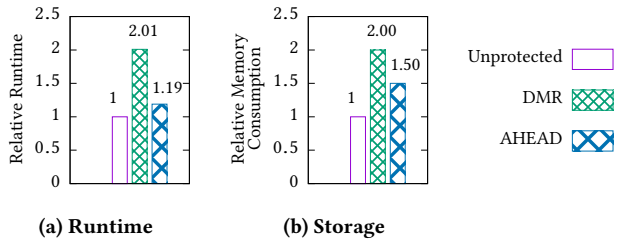
ACM Reference Format:
Till Kolditz, Dirk Habich, Wolfgang Lehner, Matthias Werner, and S.T.J. de Bruijn. 2018. AHEAD: Adaptable Data Hardening for On-the-Fly Hardware Error Detection during Database Query Processing. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3183740

1 INTRODUCTION

The key objective of database systems is to reliably manage data, where high query throughput and low query latency are core requirements [2]. To satisfy these requirements, database systems constantly adapt to novel hardware features [9, 13, 18, 35, 48, 58]. In the recent past, we have seen numerous advances, in particular with respect to *memory*, *processing elements*, and *interconnects* [12, 27, 62]. Although it has been intensively studied and commonly accepted that hardware error rates increase dramatically with the decrease of the underlying chip structures [11, 28, 74], most database system research activities neglected this fact, traditionally focusing on improving performance characteristics exploiting new data structures and efficient algorithms and leaving error detection (and error correction to some extent) to the underlying hardware. Especially for memory, silent data corruption (SDC) as a result of transient bit flips leading to faulty data is mainly detected and corrected at the DRAM and memory-controller layer [74]. However, since future hardware becomes less reliable [28, 63, 70] and error detection as well as correction by hardware becomes more expensive, this free-ride will come to an end in the near future.

The increasing hardware unreliability is already observable. For instance, repeatedly accessing one memory cell in DRAM modules causes bit flips in physically-adjacent memory cells [40, 54]. The reason for this is a hardware failure mechanism called *disturbance error* [40, 54], where electromagnetic (cell-to-cell) interference leads to bit flips. It is already known that this interference effect increases with smaller feature sizes and higher densities of transistors [40, 54]. Kim et al. [40] evaluated that all newer DRAM modules are affected and they observed one to four bit flips per 64 bit word even for error-correcting code DRAM (ECC DRAM). Other recent studies have also shown that multi-bit flips become more frequent and that the bit flip model changes at run-time due to transistor aging [40, 63]. Additionally, hardware-based protection is very challenging [28, 63, 70]. Thus, the semiconductor as well as hardware/software communities have recently experienced a shift towards mitigating these reliability issues also at higher software layers, rather than completely mitigating these issues in hardware [28, 63].

Consequently, several software-level reliability techniques have evolved, e.g., error detection using duplicated instructions [57] or



(a) Runtime **(b) Storage**

Figure 1: Relative comparison using the star-schema benchmark [68]. More details in Section 6.

software implemented fault tolerance (SWIFT) [65]. These general-purpose software techniques are usually based on data/code redundancy using dual or triple modular redundancy (DMR/TMR). However, the application of these techniques with respect to in-memory database systems causes a high overhead as illustrated in Figure 1. Obviously, software-based DMR protection requires twice as much memory capacity compared to a normal (unprotected) setting, since data must be kept twice in different main memory locations. Furthermore, every query is redundantly executed with an additional voting at the end resulting in a computational overhead slightly higher than 2x. Figure 1 highlights the average relative overheads for all 13 queries of the SSB benchmark [56, 68] with the unprotected approach as baseline.

Core Contribution. Generally, any undetected bit flip destroys the reliability objective of database systems in form of false negatives (missing tuples), false positives (tuples with invalid predicates) or inaccurate aggregates in a silent way. Since (i) general-purpose software-based protection techniques introduce too much overhead, (ii) memory systems will be significantly more error-prone in the future due to smaller chip structures, and (iii) generic hardware-level detection mechanisms will be too costly and too inflexible, there is a clear need for database-specific approaches to guarantee reliable data storage and processing without sacrificing the overall performance. In this paper, we present a novel approach called *AHEAD* for **error detection** tailored to state-of-the-art in-memory column store systems [32, 75]. As highlighted in Figure 1, our *AHEAD* approach reduces the overhead compared to DMR to a large degree for storage as well as for processing. Furthermore, multi-bit flips occurring during query processing are on-the-fly detectable, which is not the case for DMR, where errors are only detected during the voting at the end [64]. We achieve this property and the significant overhead reduction by encoding each value in a fine-grained way using a software-based error coding scheme, which allows to directly work on the encoded data representation during query processing. Moreover, we intentionally devise an **error detection**-only mechanism and leave the repair mechanism to the database system itself, e.g. by deploying a fine-grained recovery procedure. We intentionally also position our approach **orthogonal** to other coding domains like compression [1] or encryption [71], which allows a free combination of different schemes to the underlying data. To represent the intention of detecting hardware errors, we introduce new terms for encoding and decoding. We denote the process of encoding data as **data hardening**, since data is literally firmed so that corruption becomes detectable. In contrast, we denote as **data softening** the decoding of data, as it becomes vulnerable to corruption again. Furthermore, we introduce the notion of **bit flip weight** as the number of flipped bits per logical data value.

Contributions in Detail and Outline. To present our novel *AHEAD* approach in detail, we make the following contributions:

(1) Reliability concerns for future hardware have attracted major attention in the recent past. In Section 2, we give a precise problem description and define requirements for reliable data management on future hardware.
(2) We promote arithmetic codes as software-based error coding scheme and AN coding as one representative, in Section 3. In particular, we justify that AN coding has unique properties with regard to our defined requirements.
(3) Based on AN coding, we present our hardened data storage concept for state-of-the-art in-memory column stores in Section 4. Additionally, we describe the adaptation of our hardened storage for various bit flip weights and introduce performance improvements for AN code operations.
(4) Based on this hardened storage foundation, we present our on-the-fly error detection during query processing in Section 5. We introduce different error detection opportunities and determine *continuous detection* as the best solution, for which we describe essential query processing adjustments.
(5) In Sections 6 and 7, we exhaustively evaluate our *AHEAD* approach, the underlying AN coding scheme, and our developed performance improvements.

Finally, we present related work in Section 8 as well as briefly summarize the paper including a discussion of future work in Section 9.

2 HARDWARE RELIABILITY CONCERNS

Hardware components fabricated with nano-scale transistors face serious reliability issues like soft errors, aging, thermal hot spots, and process variations as a consequence of the aggressive transistor miniaturization [63]. These issues arise from multiple sources and they jeopardize the correct application execution [63]. The recently published book [63] summarizes state-of-the-art protection techniques in all hardware as well as software layers and presents new results of a large research initiative. In their work, the authors primarily target on soft errors, because soft errors are one of the most important reliability issues in the nano-era. Especially, they describe the technical backgrounds, why soft errors increase with decreasing feature sizes and higher densities of transistors [63].

Shift towards Multi-Bit Flips. The soft errors are caused due to *external influences* like energetic particle strikes, and/or *internal disruptive events* like noise transients at circuit, chip or system level, cross talks, and electromagnetic interference [63]. The soft error rate (SER) grows exponentially, because the number of transistors on the die increases exponentially [31, 63]. Consequently, multi-bit flips become much more frequent [11, 28, 63] and transistor aging leads to a changing bit flip behavior at run-time where heat stimulates this effect [28]. All hardware components are affected, but memory cells are more susceptible than logic gates [28, 30, 39].

For example, memory technologies are already much more vulnerable to electromagnetic interference effects [40, 54] (*disturbance errors*) as well as to data retention problems. A practical solution to tackle interference effects in DRAM is to increase the DRAM refresh rate, so that the probability of inducing disturbance errors

before DRAM cells get refreshed is reduced [40, 54]. However, refresh operations waste energy and degrade system performance by interfering with memory accesses [40, 54]. In addition, with increasing DRAM device capacities, the following effects with regard to DRAM refresh arise at the same time as well (*data retention problem*) [38, 37, 50, 54]. First, more refresh operations are necessary to maintain data correctly. Second, smaller DRAM capacitors require lower retention times [54]. Third, the voltage margins separating data values become smaller, and as a result the same amount of charge loss is more likely to cause multi-bit flips [38, 37, 50, 54]. Theses effects increase as DRAM capacities increase [38, 37, 50, 54].

Furthermore, emerging non-volatile memory technologies like PCM [46], STT-MRAM [44], and PRAM [86] exhibit similar and perhaps even more reliability issues [38, 37, 50, 54]. For example, PCM are based on multi-level cells (MLC) to store multiple bits in one cell which is achieved by using intermediate resistive states for storing information, in addition to the low and high resistance levels [51]. A major problem in PCM is that a time-dependent resistance drift can effect that a different value than the originally stored one will be read [51]. Furthermore, if one cell drifts to the incorrect state, other cells are also highly likely to drift in the near future. Due to this correlation, multi-bit errors changing at run-time are much more likely in MLC PCM [51]. Moreover, heat produced by writing one PCM cell can alter the value stored in many nearby cells (e.g., up to 11 cells in a 64 byteblock [34]) [51].

Protection Techniques. Hardware components usually feature protection techniques such as error correction codes (ECC) [26, 39] or hardware redundancy [19, 76]. However, these approaches are aligned to single bit flips nowadays. Thus, hardware research in two directions is necessary: (i) developing appropriate protection technique to cover new effects like *disturbance errors* and/or (ii) scaling up traditional techniques to cover multi-bit flips. Research is currently done in both directions [63]. For example, to mitigate *disturbance errors*, the inter-cell isolation is improved, but this is challenging due to ever-decreasing feature sizes and higher densities [40, 54]. Furthermore, the implementation of multi-bit error detection and correction codes in memory has been investigated [39]. However, this results in high chip area costs for storing the extra parity bits and increased calculation/checking/correction logic, whereas the overhead quickly grows with the code strength [39]. Then, reading and computing the parity bits can be a bottleneck [39].

There are a lot of hardware-oriented activities (see also Appendix B), but these activities show that hardware-based approaches are very effective, but the protection is very challenging and each technique introduces large performance, chip area, and power overheads. Furthermore, the techniques have to be implemented in a *pessimistic* way to cover the *aging* aspect leading usually to an over-provisioning. The whole is made more difficult by *Dark Silicon* [20]: billions of transistors can be put on a chip, but not all them can be used at the same time. This and the various new disruptive interference effects make the reliable hardware design and development very challenging, time consuming, and very expensive [63]. The disadvantages outweigh the advantages for hardware-based protection, so that the semiconductor as well as hardware/software communities have recently experienced a shift towards mitigating these reliability issues also at higher software layers, rather than completely mitigating these issues only in hardware [28, 63, 70].

Consequences for Database Systems. Since multi-bit flips will occur more frequently in future hardware and are not handled at the hardware layer, a major challenge is resilient query processing [7]. To tackle that, an emerging research direction will be employing protection techniques in database systems and using the available knowledge to specialize as well as to balance protection and the associated overhead. That means, appropriate solutions have to satisfy the following requirements:

- **R1:** Solutions have to *detect* and later correct (i) errors (multi-bit flips) that modify data stored in main memory, (ii) errors induced during transferring on interconnects and (iii) errors induced during computations during query processing (*detection capability*).
- **R2:** Solutions should be adaptable at run-time for different error models because the number and the rate of bit flips may vary over hardware generations or due to hardware aging effects. (*run-time adaptability*).
- **R3:** Solutions should only introduce the necessary overhead in terms of memory consumption and query runtime, which is required to detect a desired number of bit flips. That means the overhead should be as small as possible, but still provide a reliable behavior (*balanced overhead*).

In the remainder of this paper, we focus on **error detection** as a first step and present a novel approach. Our approach is tailored to state-of-the-art in-memory column stores and satisfies **R1** to **R3**.

3 SOFTWARE-BASED ERROR CODING

Our novel *AHEAD* approach is based on error coding instead of general-purpose DMR. The most important property of error codes is the detection capability, i.e. the guaranteed minimum bit flip weight, whereby the basic idea is as follows: The data word space $\mathbb{D}$ contains the original data. From a given *data word* an encoding (hardening) maps to a *code word*, which can be decoded back (softening) to the original data word in the best case without bit flips. The code domain is typically much larger than the original data domain, where only the original data words can be mapped between both domains. The code words belonging to this subset are called *valid* – valid hardened data words. All other code words are *invalid* or *corrupted*. A bit flip may change a valid code word either to an invalid one which can be detected, or to another valid one which is undetectable and known as silent data corruption (SDC).

Error codes have been widely studied in theory and applied in practice [52]. Thus, there is a large corpus of error codes available [52]. We examined a variety of these codes with regard to our three requirements **R1** to **R3**. Based on this analysis, we decided to use an arithmetic code called *AN coding*. Before we justify our decision in Section 3.2, we introduce AN coding in detail in Section 3.1. Figure 2 highlights and compares AN coding (right side) with the well-known Hamming code [26] (left) using an 8-bit example value.

3.1 AN Coding Description

AN coding is a representative of arithmetic codes[1] [4, 29], allowing only error detection. Hardened code words $c \in \mathbb{C}^{A}_{\mathbb{D}_e}$ are computed by multiplying a constant $A \in \mathbb{A}$ onto each original data word

[1]Please note that some unrelated codes for lossless lightweight data compression are also called arithmetic codes [84]. These are *not* equivalent to the codes used here.

$d \in \mathbb{D}_\Theta$ as illustrated on the right side in Figure 2:

$$c = d \cdot A \tag{1}$$

$\mathbb{A}$, $\mathbb{D}_\Theta$, and $\mathbb{C}^A_{\mathbb{D}_\Theta}$ respectively are the sets of all possible parameters A, data words d of type Θ, and code words c. The multiplication modifies the data word itself and all data is viewed as integers. As a result of the multiplication, the domain of code words $\mathbb{C}^A_{\mathbb{D}_\Theta}$ expands such that only multiples of A become valid code words, and all other integers are considered non-code words. The impact of A on the detection capability is described later. A division is required for softening:

$$d = c/A \tag{2}$$

Errors are detected by testing the remainder of the division by A, which must be zero, otherwise the code word was corrupted (c_ε), e.g., by a bit flip denoted as b:

$$c \equiv 0 \pmod{A} \tag{3}$$

$$(c_\varepsilon = c \oplus {}^2b) \not\equiv 0 \pmod{A} \tag{4}$$

A unique feature of arithmetic codes, and thus AN coding, is the ability to operate directly on hardened data by encoding the other operands, too. In particular, due to the monotony of the multiplication, the following operations on two hardened operands yield the same results as operations on unencoded data:

$$c_1 \pm c_2 = (d_1 \cdot A) \pm (d_2 \cdot A) = (d_1 \pm d_2) \cdot A \tag{5}$$

$$c_1 \circ c_2 \equiv (d_1 \cdot A) \circ (d_2 \cdot A) \overset{1/A}{\equiv} d_1 \circ d_2 \ , \circ \in \{<, \leq, =, \dots\} \tag{6}$$

Care must be taken for operations like multiplication

$$\text{OK} \quad c_1 \cdot d_2 = d_1 \cdot d_2 \cdot A, \tag{7a}$$

$$\text{BAD} \quad c_1 \cdot c_2 = (d_1 \cdot A) \cdot (d_2 \cdot A) = d_1 \cdot d_2 \cdot A^2, \tag{7b}$$

$$\text{OK} \quad c_1 \cdot c_2 \Rightarrow c_1 \cdot c_2/A = d_1 \cdot d_2 \cdot A, \tag{7c}$$

and division

$$\text{OK} \quad \frac{c_1}{d_2} = \frac{d_1 \cdot A}{d_2} = \frac{d_1}{d_2} \cdot A, \tag{8a}$$

$$\text{BAD} \quad \frac{c_1}{c_2} = \frac{d_1 \cdot A}{d_2 \cdot A} = \frac{d_1}{d_2}, \tag{8b}$$

$$\text{OK} \quad \frac{c_1}{c_2} \Rightarrow \frac{c_1}{c_2} \cdot A = \frac{d_1 \cdot A}{d_2 \cdot A} \cdot A = \frac{d_1}{d_2} \cdot A. \tag{8c}$$

While Equations (7a) and (8a) are valid operations using an encoded and an unencoded operand, Equation (7b)'s result would be invalid due to the resulting A^2 and Equation (8b) produces an unencoded result. By that, for multiplication with two hardened operands, one operand must be divided by A (Equation (7c)). For division, the result must be multiplied by A (Equation (8c)), where the correct order of the additional multiplication is crucial. The division of the two code words must take precedence to prevent an overflow.

3.2 AN Coding Justification

As shown in Figure 2, AN coding hardens the bit representation of the original value with additional bits for error detection. This applies not only to AN coding, but also to any other error code [52]. For example, the left side of Figure 2 shows the hardening using the well-known Extended Hamming code [26]. As illustrated, the Extended Hamming Code also introduces five additional parity bits

[2] $\oplus$ can be a binary XOR, OR, or AND-operation, depending on the actual error model.

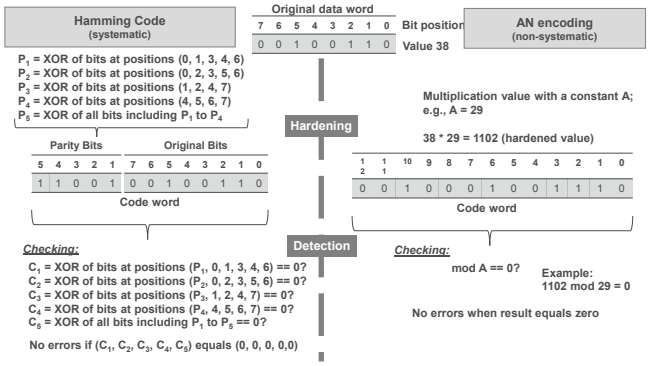


Figure 2: Illustration of Hamming and AN coding.

as AN coding in this setting, whereas four are computed using a bit mask (not shown here for brevity, but indicated in Figure 2) over the 8-bit value. The fifth parity bit P_5 is computed over all bits.

A difference between both codes is that AN coding is a representative of *non-systematic error codes*, whereas Hamming is a *systematic error code*. Systematic codes allow a separation of the original bits and additional bits. This means that the original data word is embedded as-is in the hardened representation (cf. Figure 2). In contrast, in non-systematic codes, the hardened representation does not contain the original data, so that it is not separable from the additional bits. While this makes accessing the original data faster, AN codes have a big advantage: the hardened codewords can be processed directly and each arithmetic operation also computes the additional bits, which is not the case for codes like Hamming. There, the additional bits have to be recomputed in addition to each arithmetic, introducing even more computational overhead. By that, for the database domain, codes like AN codes are preferable. Furthermore, the complete and direct processing of the hardened codewords as possible for AN codes allows the recognition of (i) errors that modify data stored in memory, (ii) errors induced during transferring on interconnects, and (iii) errors induced during computations (satisfying requirement **R1**).

In our example (Figure 2), the specific AN code with $A = 29$ and the Extended Hamming code introduce five additional bits. Both codes detect all 1- and 2-bit flips per code word, but in contrast to AN codes, Extended Hamming can also *correct* 1-bit errors. However, the correction property has a negative impact on the error detection capability for bit flips weights ≥ 2 (cf. Figure 3). The chances of *detecting* higher bit flip weights are much better for AN coding than for Hamming, because the SDC probability[3] is much lower. As depicted in Figure 3, Hamming shows a zig-zag-curve for higher bit flip weights due to the correction capabilities and this pattern holds for all Hamming codes. Many invalid code words resulting from odd-numbered bit flip weights (> 2) are mistakenly corrected to a different data word, which is not detectable. Server-grade ECC main memory uses Hamming codes and thus exhibits such behavior. This is another reason for employing AN codes for *detection* and why we concentrate on detection in this paper.

The very good detection capability combined with the easy adaptability of AN coding allows the satisfaction of the requirements

[3] The computation of these SDC probabilities is a topic of its own and described in Appendix C

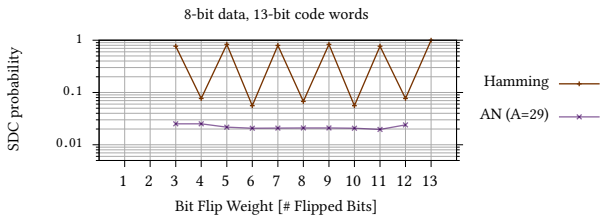


Figure 3: Comparison of SDC probabilities (lower is better) for Hamming and AN coding for our 8-bit data width example and $A = 29$. SDC probabilities for bit flip weights 1 and 2 are zero, because both codes always detect these.

R2 and **R3**. The easy adaptability of AN coding means: to detect all 3 bit flip weights or more, we only have to harden the values with a greater A. This aspect is discussed in the following section in detail. Consequently, AN coding as *error detection-only code* is a very good choice for bit flip detection in database systems allowing to balance detection capability and necessary overhead at run-time. A comparative performance evaluation is done in Section 7.

4 HARDENED DATA STORAGE

In this paper, we mainly focus on **detecting bit flips** in state-of-the-art in-memory columns by tightly integrating AN coding for data hardening. Generally, the physical data storage model of column stores significantly deviates from classical row stores [17, 32, 45, 75, 87]. Here, relational data is maintained using the decomposition storage model (DSM) [14], where each column of a table is separately stored as a fixed-width dense array [3]. To be able to reconstruct the tuples of a table, each column record is stored in the same (array) position across all columns of a table [3]. Column stores typically support a fixed set of basic data types, including integers, decimal (fixed-, or floating-point) numbers, and strings. For fixed-width data types (e.g., integer, decimal and floating-point), column stores utilize basic arrays of the respective type for the values of a column [3, 32]. For variable-width data types (e.g., strings), some kind of dictionary encoding is applied to transform them into fixed-width columns [1, 3, 6]. The simplest form constructs a dictionary for an entire column sorted on frequency, and represents values as integer positions in this dictionary [3].

4.1 AHEAD Columnar Storage Concept

Based on this, we can ascertain that column stores consist of two main data structures: (i) *data arrays* with fixed-width data types (e.g. integer and decimal numbers) and (ii) *dictionaries* for variable-width data types. Thus, each base table column is stored either by means of a single data array or by a combination of a dictionary and a data array containing fixed-width integer references to the dictionary. The decision is made based on the data type of the column. Thus, our hardened storage has to protect both structures.

Hardening Data Arrays: For data arrays with integer or decimal values, we only have to harden values and this is done using AN coding. Regarding *integer data*, this requires only multiplication with a constant factor of A. These integer values are usually compressed in column stores to reduce storage and to speedup query performance [1, 3, 21, 49, 87, 88]. We address that aspect using byte-aligned integer sizes; columns are compressed in a fixed byte-oriented way. For *decimal numbers*, the case is a bit more complex: for the sake of correctness and accuracy, database systems typically use fixed-point numbers and arithmetic instead of native floating point numbers (float / double)[4]. One possibility of representing fixed-point numbers is to split a number into powers of 100, so-called *limbs*, e.g. $1,024 = 10 * 100^1 + 24 * 100^0$. In this case, each limb fits into a single byte and the position of the decimal point is stored separately, for instance in column meta data. In general, the limbs can, of course, be larger. Using this representation, there are two options for hardening a fixed-point number: (1) harden each limb, or (2) harden the number as a whole. The former approach requires adapting the algorithms to work on larger limbs, as each limb becomes a code word on its own. Using the latter approach allows to leave the algorithms unchanged, but unfortunately, deriving the detection capabilities for large arbitrary data widths is very expensive[5]. Consequently, only the former approach is feasible.

Hardening Dictionaries: Dictionaries are usually realized using index structures to efficiently encode and decode [6]. In contrast to data arrays, not only data values must be hardened, but also necessary pointers within the index structures. To tackle that issue, we proposed various techniques to harden B-Trees [43]. As we have shown, hardening pointer-intensive structures pose their own challenges and we refer to this solution for hardening dictionaries. Moreover, for dictionaries of integer data, AN hardening can be applied on the dictionary entries. The corresponding column contains fixed-width, AN hardened integer references to the dictionary.

UDI-Operations: Our hardening approach is orthogonal to UDI-operations (Update, Delete and Insert) and does not affect them. New and modified data is usually appended at the end of a column, and hardening such data is trivial – hardened data simply has to be inserted.

4.2 AHEAD Adaptability

AN coding has only one parameter A having an impact on the error detection rate as well as the necessary storage overhead. Generally, there are arbitrarily many As to choose from and we now clarify which one to actually use. As it turns out, each A has different error detection properties with regard to the data width. This is why $\mathbb{C}^A_{\mathbb{D}_\Theta}$ depends on both A and $\mathbb{D}_\Theta$. It follows that for *any* distinct data width, which the database supports, the error detection capabilities for a large set of As must be computed. To compute the detection properties of an A, we have to determine the Hamming distance d_H of all pairs of code words, i.e., the difference in the number of bits of any two code words. In other words, this corresponds to a bit population count of all error patterns, which leads from any valid code word to any other valid one. In the coding domain this is known as a code's *distance distribution*. Next, a histogram over that distribution must be built and from that we derive the *minimum Hamming distance* $d_{H,\min}$. This is the smallest Hamming distance $\neq 0$ in the distance distribution and any code is guaranteed to detect *at least* all error patterns with up to $d_{H,\min} - 1$ flipped bits.

[4] We would like to point to the article by Thomas Neumann providing a discussion and background information on this issue [55]. In essence, rounding and precision loss problems of native floating-point numbers and operations are usually unacceptable for database systems. This is why these systems employ fixed-point arithmetic and there exist several libraries providing a variety of mathematical operators [23, 24, 66].
[5] See Appendix C for details.

Table 1: Super As for detecting a guaranteed minimum bit flip weight (min bfw), excerpt from $|D| \in \{1, \ldots, 32\}$. Numbers are: $A/|A|/|\mathbb{C}^A_{\mathbb{D}_\Theta}|$. *=derived by approximation, bold=prime, tbc=to be computed.

min bfw	$\lvert\mathbb{D}_\Theta\rvert$ 8	16	24	32
1	**3**/2/10	**3**/2/18	**3**/2/26	**3**/2/34
2	**29**/5/13	**61**/6/22	**61**/6/30	125/7/39
3	**233**/8/16	**463**/9/27	981/10/34	**881**/10/42
4	1,939/11/19	7,785/13/29	15,993/14/38	16, 041*/14/46
5	**13,963**/14/22	63,877/16/32	tbc	tbc
6	55,831/16/24	tbc	tbc	tbc

As described in Appendix C, obtaining the distance distribution requires a brute force approach. For each combination of bit widths $|\mathbb{D}_\Theta|$[6] and $|A|$, there exists at least one "super A" having optimal error detection capabilities among all other As for that combination. For that, our optimality criterion is that that a "super A" has (1) highest $d_{H,\min}$, (2) lowest bit width $|A|$, and (3) lowest first non-zero histogram value in its category, i.e., depending on the minimum bit flip weight (`min bfw`) and $|\mathbb{D}_\Theta|$. Table 1 lists an extract of "super As" for different numbers of minimum bit flip weights and different data widths $|\mathbb{D}_\Theta| \in \{8, 16, 24, 32\}$. This table also confirms our example from Section 3 where we used A=29 for 8-bit data and a minimum bit flip weight of two. As depicted, we require five additional bits for the hardening. If we want to increase the minimum bit flip weight to 3, we only have to use $A = 233$ resulting in a code word width of 16. In this case, the data overhead increases from 62.5% (13 bit code word width) to 100% (16 bit code word width for 8 bit data). Table 3 in the appendix lists all super As which we obtained until now, for all $1 \leq |\mathbb{D}| \leq 32$ and $1 \leq |A| \leq 16$. The determination of the "super As" is extremely compute-intensive, because only a brute force approach is possible. Some values are currently only approximated[7], which is only an intermediate solution. Computations for $1 \leq |\mathbb{D}| \leq 32$ and $1 \leq |A| \leq 16$ took 2700 GPU hours in total on an nVidia Tesla K80 GPU cluster, including the approximations. Time complexity is in $O(2^{2 \cdot |\mathbb{D}|})$ and doubles with every additional code bit. Our approximation is configurable through parameter M and reduces runtimes by $2^{|\mathbb{D}|}/M$ and we used $M = 1001$. For instance, runtimes are reduced by more than 5 orders of magnitude for $|\mathbb{D}| = 27$. The maximal relative error is below 1% for code lengths which we could verify exhaustively.

To summarize, we have calculated the "super As" not only for byte-aligned data widths, but also for any data widths between 1 and 32. The "super As" adhere to our optimality criterion and are the smallest ones for detecting all bit flips up to a minimum bit flip weight. As we can see in Table 1: (1) equal or different As are optimal for the same minimum bit flip weight and varying $|\mathbb{D}_\Theta|$; (2) for increasing $|\mathbb{D}_\Theta|$, we typically need larger $|A|$s to achieve the same detection capability; and (3) not all super As are prime numbers. Now, we are able to use this information for a balanced data hardening with regard to a specific hardware error model (bit flip weights and rates) and data width. Additionally, data can be re-hardened at run-time with different As. Thus, the requirements **R2** and **R3** are adequately addressed from the storage perspective.

[6]We use the following notation of || for bit widths in this paper.
[7]The remaining correct values are still determined and will be made public on our GitHub project page https://brics-db.github.io/ (see Appendix D)

4.3 AHEAD Performance Improvements

Our third requirement **R3** is that the approach introduces as little overhead as necessary. Up to now, we have shown that the memory overhead can be adjusted according to the data width and the required error detection capabilities. Unfortunately, error detection and softening/decoding are based on division and modulo computations, which are expensive even on modern server-grade processors (cf. Section 7). We now show how to circumvent both operations.

Faster Softening: Processors' arithmetic logic units (ALUs) implicitly do computations in a residue class ring (RCR) modulo the power of two to the native register width (e.g. 8, 16, 32, 64 bits). In such an RCR, the division by any odd A can be replaced by multiplication with the *multiplicative inverse* A^{-1}. This cannot be done automatically by the compiler since the A will only be known at run-time and the code width may differ from the native register widths. Now, decoding becomes:

$$c/A \equiv c \cdot A^{-1} \equiv (d \cdot A) \cdot A^{-1} \equiv d \mod 2^{|\mathbb{C}^A_{\mathbb{D}_\Theta}|}. \tag{9}$$

The inverse can be computed with the Extended Euclidean algorithm for the RCR modulo $2^{|\mathbb{C}^A_{\mathbb{D}_\Theta}|}$. When working with code word widths different from the processor's native register widths, the result must be masked, i.e. AND-ed with a bit mask having the appropriate $|\mathbb{C}^A_{\mathbb{D}_\Theta}|$ least significant bits set to one. This is because there may be ones in the remaining most significant bits from the multiplication. Using the inverse has several advantages:

(1) Only odd numbers are coprime to any 2^n ($n \in \mathbb{N}$) and thus have a multiplicative inverse. Consequently, it is required to use *only odd* As.
(2) Using the inverse relieves Equation (7c) from the division.
(3) The inverse enables more efficient reencoding from one code word $c_1 = d \cdot A_1$ into another $c_2 = d \cdot A_2$, by multiplying with the factor $A^* = A_1^{-1} \cdot A_2$:

$$c_1 \cdot A^* = (d \cdot A_1) \cdot (A_1^{-1} \cdot A_2) = d \cdot A_2 = c_2 \tag{10}$$

The product $(A_1^{-1} \cdot A_2)$ is a constant factor and needs to be calculated only once, especially when reencoding multiple values.

Faster Error Detection: Using the multiplicative inverse allows to get rid of the modulo operator for error detection, too. For that, as in [29], we must know the largest and smallest encodable data word:

$$d_{\max} = \max(\mathbb{D}_\Theta),$$
$$d_{\min} = \min(\mathbb{D}_\Theta),$$

where the latter is required for signed integers, only. Since computations on code words must be done on register widths $\geq |\mathbb{C}^A_{\mathbb{D}_\Theta}|$, it follows that:

$$|c \cdot A^{-1}| = |d^*| = |c| > |d| \quad , d^* = d. \tag{11}$$

I.e., when decoding, the resulting data word d^* is computed in the larger RCR (modulo $2^{|\mathbb{C}^A_{\mathbb{D}_\Theta}|}$) than the original data d requires (modulo $2^{|\mathbb{D}_\Theta|}$). This becomes very useful, because we discovered

that in this case it holds that:

$$d^* > d_{\max} \rightarrow d^* = (d \oplus b) \tag{12}$$

$$d^* < d_{\min} \rightarrow d^* = (d \oplus b) \tag{13}$$

where b is an arbitrary, *detectable* bit flip of any weight and d^* was decoded from a corrupted code word. This goes further than in [29], because we found that it holds for *any* detectable error. For signed integers, the binary representation contains ones in the $|A|$ most significant bits (MSBs) where there should be zeros after the multiplication with the inverse. Likewise, the same holds for negative integers, but now there are *zeros* in the $|A|$ MSBs, where there should only be ones. For unsigned integers, the first test suffices, while for signed integers both tests must be conducted. Consider the following example for signed integers (sint) with $|\mathbb{D}_{\text{sint}}| = |A| = 8 \Rightarrow |\mathbb{C}^A_{\mathbb{D}_{\text{sint}}}| = 16, d = 5, A = 233$, and $A^{-1} = 55,129$ (unsigned) $= -10,407$ (signed):

		overflow	$\lvert A\rvert$	$\lvert\mathbb{D}_{\text{sint}}\rvert$
1)	$5 \cdot A = 1,165$	$= \ldots 0000$	0000 0100	$1000\ 1101_2$
2)	$1,165 \cdot A^{-1}$	$= \ldots 0100$	0000 0000	$0000\ 0101_2$
3)	$+1: 1,166 \cdot A^{-1}$	$= \ldots 0100$	1101 0111	$0101\ 1110_2$
4)	$-1: 1,164 \cdot A^{-1}$	$= \ldots 0011$	0010 1000	$1010\ 1100_2$

The first line shows the encoding, with the result in binary representation. The second line is the decoding of the valid code word, with no zeros in the $|A|$ MSBs. Lines 3 and 4 show decoding of altered code words, where $1,165 + 1$ and $1,165 - 1$ represent a double and single bit flip in the LSB(s), respectively. The overflow column shows that computing with non-register widths requires masking for decoding. For the signed case, line 3 triggers the test from Equation (13) and line 4 the one from Equation (12). Although we cannot prove this in general[8], we could confirm this for all odd As with $2 \leq |A| \leq 16$, signed integers with $1 \leq |\mathbb{D}_{\text{sint}}| \leq 24$, and $|\mathbb{C}^A_{\mathbb{D}_{\text{sint}}}| = |A| + |\mathbb{D}_{\text{sint}}|$. We validated Equations (12) and (13) using an exhaustive test over all possible code words[9]. From Equations (12) and (13) it follows that testing an AN-encoded data word for errors is achieved by *first decoding* it and *then comparing* it with the largest (and smallest) encodable data word for unsigned (signed) integers. For error testing and decoding, Hoffmann et al. [29] use a comparable approach for their voter, but they require two comparisons and the modulo and division operators. In contrast, our approach requires one multiplication and one or two comparisons.

5 ON-THE-FLY ERROR DETECTION

On-the-fly error detection during query processing becomes now possible for both state-of-the-art processing models of column-at-a-time [3, 32] and vector-at-a-time [87] with our hardened storage concept. There are two reasons: (i) the column structure is unchanged, only the data width is increased and (ii) the values are multiplied by A and can thus be processed as before.

5.1 Error Detection Opportunities

Basically, there are three opportunities for on-the-fly error detection as shown in Figure 4, whereby Figure 4a represents an example query execution plan (QEP) without any hardened data. In this QEP, data from two columns R and S is filtered, the results are joined, and finally grouped by column R. We use this QEP, to describe our three detection possibilities during query processing.

Early Onetime Detection: As a first possibility, error detection happens only once, when touching data the very first time in the base columns (Figure 4b). For that, we introduce the *detect-and-decode* operator Δ taking as input a column containing hardened data and outputing a column containing the decoded data. Δ is put before any other operator in the QEP. The physical realization of Δ is based on a regular column scan-operator conducting detection and decoding based on the formulas presented in Section 4.3. However, the disadvantage is that bit flips during the remaining query processing are not detected.

Late Onetime Detection: Second, since AN coding allows arbitrary operations directly on hardened data, detection may take place in a late stage of a QEP. For instance, a hardened column can be filtered just by encoding the filter predicate with the same A the input columns are hardened with. Thus, our *detect-and-decode* Δ can be placed in the QEP at a point, where there are still sufficiently many tuples left for bit flip detection, e.g. before a group-by as in Figure 4c. Operators taking multiple inputs, like joins, may encounter differently encoded inputs so that Δ must be placed before these (Figure 4d, Section 5.2 provides further details). However, the disadvantage is that bit flips are only detected at a single point and errors are propagated through the QEP.

Continuous Detection: Thus, we have to integrate bit flip detection into each and every physical operator (Figure 4e). This is conceptually the best solution, because bit flips are quickly detected and Δ becomes superfluous. By that, each and every value is checked for bit flips in the columns of base tables and intermediate results.

5.2 Adjustments for Continuous Detection

We now focus on necessary adjustments to the query processing to realize the continuous error detection approach.

Error Detection Primitives for Physical Operators: Each physical query operator has to be adjusted to include appropriate AN coding detection primitives. We do this with a vector storing the array position of corrupted values per column. Of course, this error vector can also be affected by bit flips, so we harden it with AN coding, too. Further adjustments are exemplarily described for a `filter scan` operator on signed integers, as shown in Algorithm 1. For reasons of better traceability, the algorithm description is conceptually based on MonetDB primitives using their BAT data structures for columns [8]. The operator takes two inputs: (i) a single hardened input BAT and (ii) a hardened predicate (*pred**). To provide a full-fledged example, assume that (i) both head and tail of the input BAT are non-virtual and encoded with different As and (ii) the operator returns a hardened output BAT and two error vectors. First, the operator initializes the error vectors (lines 1 and 2). Variable `pos` (lines 3 and 15) is used to keep track of the position in the input BAT, so that we are able to store the error position within the input BAT. When iterating over head and tail they are first decoded (lines 5 and 6) and then tested according to Equations (12) and (13) (lines 7 and 10). Upon detection of an error, the hardened position is stored in the appropriate error vector (lines 8 and 11). If the tail value is assumed to be valid, the filter is

[8] A proof is very difficult due to the convolution of the multiplication.
[9] On a 64-core AMD Opteron 6274 server it took almost 50K CPU hours in total.

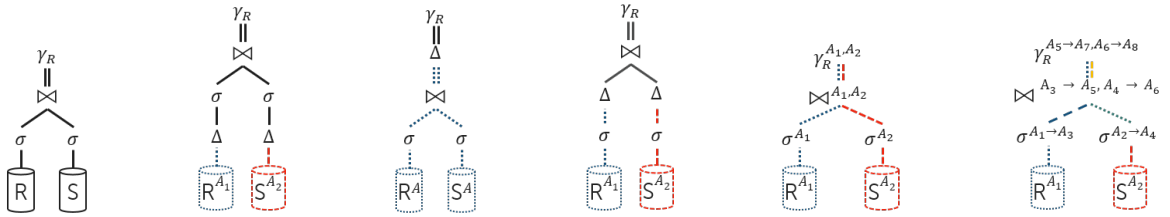


Figure 4: Illustration of the query processing variants. Δ is a dedicated check-and-decode operator. Different As are indicated via superscripts at the columns and operators, and via different colors and line decorations. Reencoding is indicated as $A_1 \rightarrow A_2$.

Algorithm 1 `Filter scan` for continuous error detection, having head and tail hardened with different As and reencoding its output. * denotes hardened values and BATs. Subscript $_h$ and $_t$ denote head and tail, respectively.

Input: B^*_{in} ▷ hardened input BAT
Input: $A_h, A_h^{-1}, A_t, A_t^{-1}$ ▷ AN coding parameters
Input: A'_h, A'_t ▷ Reencoding parameters
Input: A_{pos} ▷ Error position hardening parameter
Input: $d_{h,\max}, d_{t,\max}$ ▷ largest unencoded values for B^*_{in}
Input: $d_{h,\min}, d_{t,\min}$ ▷ smallest unencoded value for B^*_{in}
Input: $\circ \in \{<, \leq, =, \neq, \geq, >\}$ ▷ comparison operator
Input: $pred^*$ ▷ predicate encoded with A_t
Output: B^*_{out} ▷ result BAT (same A as B^*_{in})
Output: v_h, v_t ▷ error bit vectors for **h**ead and **t**ail
1: $v_h.initialize()$ ▷ allocate memory for v_h
2: $v_t.initialize()$ ▷ allocate memory for v_t
3: pos ← 0
4: **for each** (head*, tail*) ∈ B^*_{in} **do**
5: $h \leftarrow \text{head}^* * A_h^{-1}$
6: $t \leftarrow \text{tail}^* * A_t^{-1}$
7: **if** $h < d_{h,\min}$ **or** $h > d_{h,\max}$ **then**
8: $v_h.append(pos * A_{\text{pos}})$
9: **end if**
10: **if** $t < d_{t,\min}$ **or** $t > d_{t,\max}$ **then**
11: $v_t.append(pos * A_{\text{pos}})$
12: **else if** tail* $\circ$ $pred^*$ **then**
13: append $(h * A'_h, t * A'_t)$ to B^*_{out}
14: **end if**
15: pos ← pos +1
16: **end for**

evaluated (line 12) and if it matches, the hardened column value is appended to the output BAT (line 13). Likewise, this pattern can be applied to all other physical operators, whereas the number of error vectors varies depending on the number of input columns.

Physical Operations with Different As: Table 1 shows that there are different super As for different data widths. Consequently, base columns will be hardened with different As. In Figure 4 different As are indicated by superscripts at data and operators and by different line types and colors between operators. Columns with different As are a challenge for binary operations like joins (see Figure 4e). There, for different As it holds that $(d \cdot A_1) \neq (d \cdot A_2)$, $A_1 \neq A_2$ and an equi-join would at most produce false positives, when $d_1 \cdot A_1 = d_2 \cdot A_2$, $d_1 \neq d_2$. To produce correct results, binary operators have to adapt one of the inputs (hardened with A_1) to match the other's A_2 by multiplying the factor $A_1^{-1} \cdot A_2$ onto each hardened value of the first input column, or $A_1 \cdot A_2^{-1}$ onto the second. Optionally, the decoded value, which is computed for error detection and resides in a CPU register, could be multiplied by A_2 only, as well. As a side effect, each operator may well harden or soften the data on-the-fly (Figure 4f), called reencoding. This can, e.g., reduce the data width for intermediate results or prepare the data for the following operator (cooperative operators as introduced in [41]). Consequently, it allows to adjust space requirements against reliability, i.e., multi-bit flip detection capabilities. Reencoding is actually the same as the adaptation described for the join operator. In Figure 4f, each operator reencodes its input with a new A, indicated as $A_x \rightarrow A_y$. Generally, this reencoding is a very interesting property as will be shown in the evaluation.

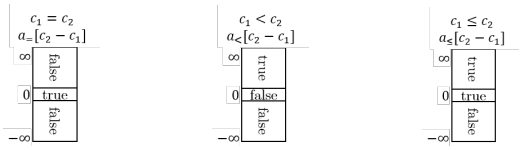
Handling of Intermediate Results: Because all operations can directly work on hardened data and each operator passes its hardened results to the next operator, the intermediate results are automatically hardened. There is only one special feature in column stores, namely, virtual IDs can sometimes be materialized during the query processing [3, 32]. In order to detect errors during the further processing of these virtual IDs, these IDs are automatically hardened during their generation. Differently encoded IDs can be handled as indicated above for the join.

Bit Flip Detection in ALU Operations: As described in Section 3, AN coding allows to detect errors in certain operations of a CPU's ALU (arithmetic logic unit), which performs arithmetic, comparison and logic operations. For arithmetic operations like $+, -, /, \ldots$ (cf. Equation (5) and eqs. (7a), (7c), (8a) and (8c)), the result can be immediately checked by multiplying with the respective A's inverse. The checks must be anchored in the corresponding physical operators, as already described.

Error detection for comparison operations like $<, \leq, =, \ldots$ (cf. Equation (6)) is much more challenging, since comparisons are required for the detection itself. We could use a technique similar to branch-avoidance by replacing the comparison with an array access, as depicted in Figure 5. Suppose that for each comparison operator there is an infinitely large array of boolean values:

$$a_{\text{Op}}[i] \in \{\top, \bot\},\ \text{Op} \in \{<, \leq, \ldots\},\ i \in \{-\infty \ldots 0 \ldots \infty\}.$$

The difference of the two operands yields the position in the array, where we find the boolean value representing the result of the



(a) Equal **(b) Less-Than** **(c) Less-or-Equal**

Figure 5: Replacing comparison by access to a boolean array.

original comparison operation. For instance, Figure 5a illustrates the array contents for the equality comparison. There, we compute the difference of the two operands $c_2 - c_1$ and the array contains only one `true` value at position zero, since $c_2 = c_1 \Leftrightarrow c_2 - c_1 = 0$. Likewise, for '<' comparison (Figure 5b), we find `true` only for $i > 0$ and for '≤' `true` is at all $i \geq 0$. However, infinite arrays can not be stored in memory and their contents would have to be verified at runtime as well.

Consequently, errors in logic operations ($\&, \|, \oplus, \neg$) cannot be detected solely by AN coding and bit operations like AND-ing, OR-ing, XOR-ing, or inverting all bits produce *invalid* code words. By that, these operations must be protected in another way. Therefore, we assume *reliable* comparison and logic operations. Both areas must be considered separately in future work.

Vectorization: The array-like hardened storage of columns, where only values of the very same data type are stored, allows to use vectorization (SIMD) instructions [88, 83]. Our improved AN coding requires only multiplication and comparison, which are well supported as SIMD instructions in virtually all modern processors. Consequently, for code word widths which fit into native SIMD registers, our *AHEAD* approach can be well combined with vectorizable physical operators like filter or aggregation. Willhalm et al. show how data with bit widths from 1 to 32 bits is aligned in SIMD registers to be processable, even with complex predicates and with special optimizations for each bit case [83]. After their alignment step, multiplication and comparison instructions for AN coding operations can be included. There are, however, some peculiarities with SIMD operations. As of now, instruction sets like Streaming SIMD Extensions (all variants until SSE4.2) and Advanced Vector Extensions 1 & 2 do not support unsigned integer comparison, which must be emulated with 2 operations. For instance, for the unsigned "$>$" comparison in SSE4.2, instead of a single

$v_> =$ `_mm_cmpgt_epu(vector(`$d_0, d_1, \ldots$`), vector(`$d_{\max}$`))`,

the following is required:

$v_{\text{tmp}} =$ `_mm_max_epu(vector(`$d_0, d_1, \ldots$`), vector(`$d_{\max} + 1$`))`

$v_> =$ `_mm_cmpeq_epi(`v_{tmp}`, vector(`$d_0, d_1, \ldots$`)`

Afterwards, the result of the comparison can be translated into an integer using one of the `_mm_movemask` instruction variants. AVX-512 is supposed to provide the comparisons natively and some operations even immediately return an appropriate mask, so instead of three instructions, a single one suffices – at least from a source code perspective.

6 END-TO-END EVALUATION

We evaluate our *AHEAD* approach using the analytical SSB benchmark [56]. All experiments are conducted without error induction, because the conditional SDC probabilities are known (cf. Section 4). Our system was equipped with one Intel® Core™ i7 6820HK CPU (@ 2.70GHz), and 16 GB of DDR3 main memory (@ 2133MHz) running 64-bit Ubuntu 16.04.3 LTS using GCC 7.1.0.

6.1 Implementation

We implemented a self-contained *AHEAD* prototype using a column-at-a-time execution model[10], due to the following: (1) We wanted to have both protected and unprotected data to measure the appropriate runtimes in a uniform way. This requires to have operators for both available simultaneously, which would require to reimplement all physical operators in an existing DBMS. (2) We want to first show that our approach is feasible, without any side effects, so that it can then be integrated in an existing system. Our *AHEAD* prototype is completely written in C++ using well-known column store concepts [3] and supports both unprotected and hardened query processing in a unified way for comparability. For that, a separate type system allows to distinguish data types and the actual width of a type is adjusted through a mere `typedef`. Our current prototype differentiates between 8-, 16-, 32-, and 64-bit data types, i.e. we do byte-level compression on native CPU register granularity. We call the types `tinyint`, `shortint`, `int`, and `bigint`, respectively. The hardened variants (`restiny`, `resshort`, `resint`, and `resbig`, respectively) are mapped to the next available native integer width, i.e. 16, 32, 64, and again 64 bits. For `tinyint`, this allows *A*s up to a width of 8-bits, for the others *A*s up to 16-bit and `bigint` is limited to 48-bits of actual data[11]. For strings, we use a separate data heap and the data column contains pointers to the actual string values. Our current implementation does not use dictionary compression. The physical query operators support all these data types through template programming to ease the data type specialization. Furthermore, for the vectorized operators, the template programming helps to specialize only those details of the SSE operators which require calls to specific intrinsics. For instance, the algorithm skeletons are the same for all integer types, but e.g. multiplication requires different intrinsics for 16-bit (`_mm_mullo_epi16`, `pmullw`) or 32-bit (`_mm_mullo_epi32`, `pmulld`) data, just to name two examples. We implemented both, scalar and SSE4.2 query operators, while Hash-join and Group-by use Google's `dense_hash_map`[12] for performance reasons. All of the 13 SSB queries are manually written, guided by the query explanation output from MonetDB [10]. We chose single-threaded execution to avoid any side-effects and to precisely measure the overhead.

6.2 SSB Runtime Performance

We compare our *AHEAD* approach with the *Unprotected* baseline and dual modular redundancy (DMR). In the *Unprotected* baseline, data is always compressed on a byte-level based on the column characteristics. *DMR* uses the *Unprotected* setting and replicates all data in main memory, executes each query twice sequentially, and afterwards a voter compares both results. Our *AHEAD* approach hardens each column using the largest currently known *A* for the corresponding column data width from Table 1. Thus, compared

[10] The prototype is available on GitHub https://brics-db.github.io/, see Appendix D
[11] 128-bit integer implementations like the Boost C++ library could be used to support larger integer widths, using a simple `typedef` change for scalar code.
[12] https://github.com/sparsehash

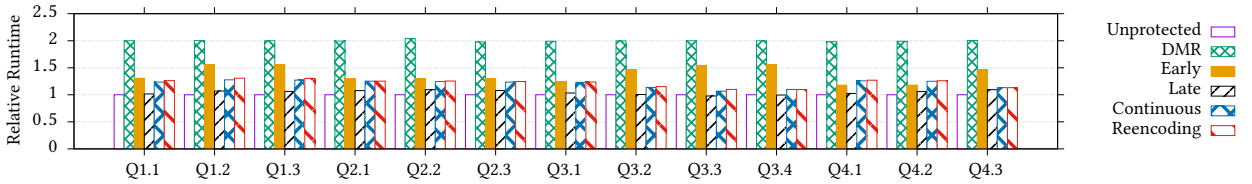


Figure 6: Relative SSB runtimes for vectorized (SSE4.2) execution (average over all scale factors).

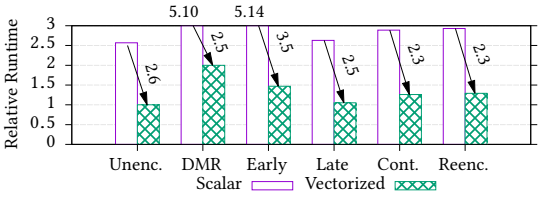
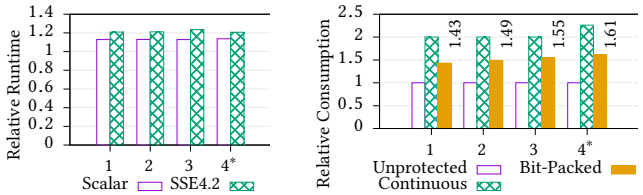


Figure 7: Scalar against vectorized runtimes (SSB 1.1 to 1.3).



(a) Runtime **(b) Storage**

Figure 8: SSB query 1.1 runtime and storage comparison for *Continuous* with different min bfw's (x axis). 4*: we increased `restiny` size to 32 bits to allow $|A| > 8$.

to *Unprotected* setting, *AHEAD* increases the data width of each column to the next byte level. For all approaches, we measured all 13 SSB queries for scalar and vectorized (SSE4.2) execution and we varied the SSB scale factor from 1 to 10. Each single experiment ran 10 times. Figure 6 shows vectorized (SSE4.2) runtimes relative to the *Unprotected* baseline. *DMR* results in the expected runtime overhead of about 100%, because each query is executed twice. For our *AHEAD* detection schemes, we can draw the following conclusions:

Early Detection has a very high initial overhead of the Δ operator leading to an overhead between 64% and 185% for scalar execution (cf. Figure 11 in Appendix A). With vectorization, early detection is much faster, because Δ benefits so much more than the other operators, and the overhead is reduced to 18% to 56%.

Late Detection executes Δ only on hardened data arrays prior to the aggregation or grouping and newly created intermediates like IDs are never hardened. Consequently, it checks much less values and errors are detected only very late in the query, resulting in overheads of at most 10% for both scalar and SSE4.2 execution. Errors might add up to valid code words, so that less are detected.

Continuous and Reencoding use the same query plan as the baseline but with AN-coding-aware operators, where the latter reencode each operator's output. As we assume no particular error rate or bit flip weight distribution, our policy for the new A for *Reencoding* is to decrease the bit width of A by 1 for each input. For that, we use an extended version of Table 1. This is meant to simply show the feasibility of reencoding each operator output. The runtime of *Continuous Detection* lies between Early and Late with about 10% to 26% (scalar) and 7% to 28% (vectorized) overhead. This is due to the tight integration of error detection in each operator. *Reencoding* adds virtually no overhead to *Continuous* with overheads of 10% to 27% (scalar) and 10% to 31% (SSE4.2).

6.3 Scalar vs. Vectorized Execution

Having compared the detection variants per query, we now show in more detail the impact of the vectorization. We use the average over SSB queries 1.1 to 1.3 because they filter on small integer data and there the impact of vectorization is greatest. Vectorized *Unprotected* serves as baseline. Figure 7 shows that all variants benefit greatly from SSE4.2 vectorization for queries 1.1 to 1.3. The speedup factor is shown at the arrows for each variant, as well. For *Early* it is larger because Δ benefits very much. Note that the speedups for *Continous* and *Reencode* are only slightly smaller than for *Unprotected*, although they operate on half as many code words per SIMD instruction. Consequently, our AHEAD approach scales well using vectorization.

6.4 Influence of Bit Flip Weight

Up to now, we used the largest A for data hardening in our experiments. We will now investigate the influence of different As on the runtime and storage overhead of the *Continuous* variant. For each hardened data type, we vary the A of all base and intermediate columns to the smallest one for guaranteed minimum bit flip weights (*min bfw*) 1 to 4 from Table 1. We chose this scenario, because Kim et al. [40] observed one to four bit flips in all newer DRAM modules. For the case of 4, we let the `restiny` datatype be 32 bits wide to allow $|A| > 8$. Due to our type system, this is a simple `typedef` change. Figure 8a shows the runtimes as in Figure 6, while Figure 8b shows the storage overhead. Runtimes differ only slightly for both scalar and SSE4.2 execution. With *Continuous*, the storage consumption doubles for *min bfw* 1 to 3, because we only work on native register widths. For *min bfw* 4, the storage overhead is slightly higher (126%), since `restiny` is twice as big. This shows the limitation of our current byte-oriented compression. Willhalm et al. showed how to store data in a bit-packed fashion and how to evaluate even complex filter predicates in a vectorized manner [82, 83]. To overcome our current implementation limitation, we could use their approach. To show the benefit, we projected the storage consumption for *bit-packed* compression in Figure 8b as well, which displays potentials for reducing the memory overhead of our *AHEAD* approach. Since bit-packing reduces the load on the memory subsystem, this *might* improve query runtimes, too, but that remains to be seen.

7 MIRCO BENCHMARKS

We also conducted micro benchmarks to support our error code decision from a performance perspective.

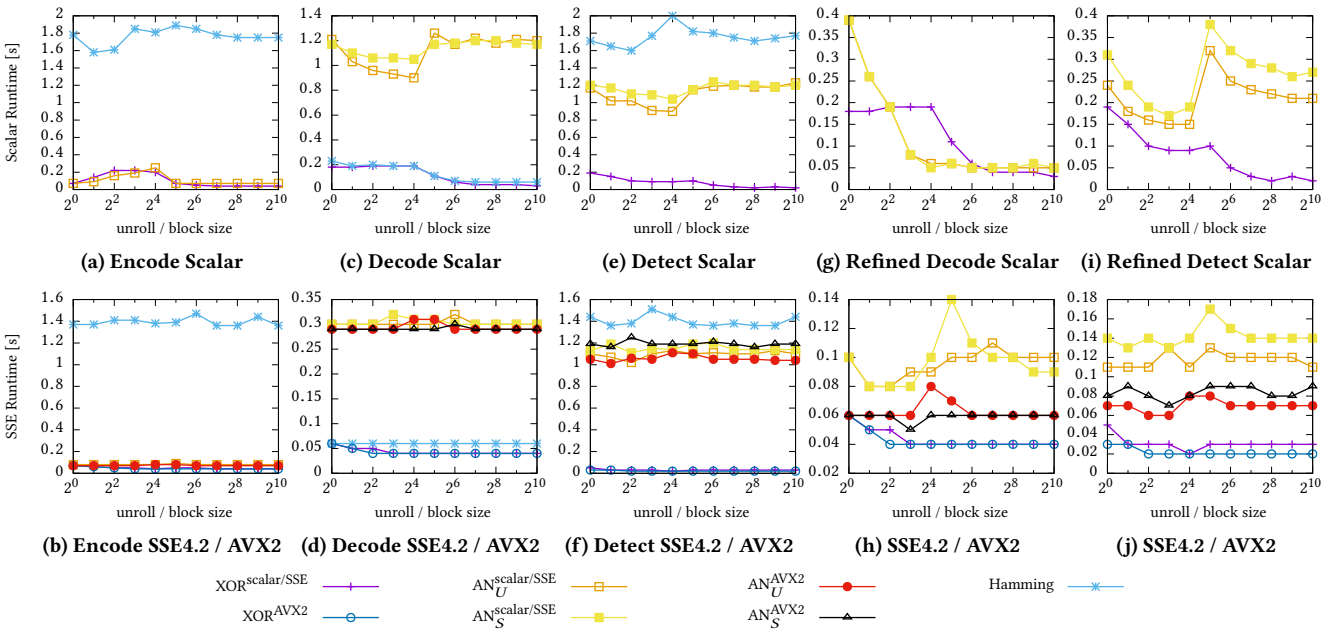


Figure 9: Micro Benchmarks – runtimes for scalar, SSE4.2 and AVX2 code functions. For XOR the block size is varied, while for Hamming and AN a loop unroll factor is varied. AN_U = unsigned, AN_S = signed.

7.1 Error Code Evaluation

In our first micro benchmark series, we compared AN coding with other well-known und heavily applied coding schemes to show the advantages of AN coding for software-based error detection. In detail, we consider Hamming codes as presented in Section 3 and checksums, both *systematic error codes* where data bits and additional, redundant bits are separable. *Checksums* add a small-sized value derived from an arbitrary data block forming a hardened data block allowing limited error detection. Nowadays, the term checksum is also used when hash functions are applied. There exists a multitude of algorithms with varying complexity and hardware support, e.g. parity bits, parity words, Message-Digest Algorithms (e.g. MD5) [67, 73] or cyclic redundancy checks (e.g. CRC32) [59, 73]. In the case of parity bits (words), the data bits (words) are summed up using the binary exclusive or operation (XOR, $\oplus$). The size of the resulting checksum can be arbitrary, but is usually either a single bit or aligned to machine words, respectively, for the sake of performance. In the following, we restrict our discussion to XOR checksums, since they are one of the most simple types of checksums. Furthermore, both Hamming code and XOR checksum can be vectorized [53, 80] which is important for performance.

In detail, we compared runtime overheads for the following functions: *hardening*, *decoding* and *error detection*. For this, we implemented these functions for all three error codes using C++ template-meta programming to let the compiler unroll the code. We always implemented scalar and SSE4.2 variants, and AVX2 for AN and XOR. In the experiments, we processed about 250 Million 16-bit integers depending on either the blocksize (XOR) or a loop unroll factor (AN, Hamming). The blocksize indicates over how many values a checksum is computed, whereas Hamming and AN coding is applied on each single value. Furthermore, our vectorized algorithms take into account that the number of values may not be aligned to the blocksize/unroll factor.

Hardening Overhead. Figures 9a and 9b show the runtimes for data hardening/encoding. As we can see, Hamming coding is more than an order of magnitude slower than XOR and AN coding. For AN coding, signed and unsigned are equal.

Decoding Overhead. Figures 9c and 9d show that decoding is straightforward for Hamming and XOR, as the redundant bits are easily separable from the original data bits. The original AN coding, however, requires expensive integer division, so that both sequential and vectorized variants are more than an order of magnitude slower than the other two.

Detection Overhead. For bit flip detection, Hamming and XOR have to recompute the redundant bits and compare them against those retrieved from memory. This is basically the same as encoding with additional comparisons. Figures 9e and 9f show that XOR detection is the fastest. Original AN coding shows poor performance due to the expensive modulo operator, for which no SSE or AVX SIMD instructions exist. Hamming is again more than an order of magnitude slower than XOR, but since population count computation can be vectorized, it comes closer to AN coding.

Conclusion. From a performance perspective, the original AN coding approach does not perform very well compared to XOR checksums. Thus, we introduced our improvements in Section 4.3.

7.2 Evaluation of AN Coding Improvements

In the second micro benchmark series, we evaluated our AN coding improvements for faster softening and detection. Figures 9g and 9h show the decoding performance and Figures 9i and 9j depict the detection performance. Comparing these improvements with XOR, we see great improvements over the original AN coding

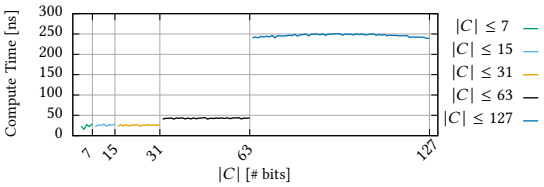


Figure 10: Runtimes for computing multiplicative inverses.

variant. The improved AN coding is much closer to the XOR performance. For these improvements, we require the multiplicative inverse for a given A. As illustrated in Figure 10, the calculation can be done on-the-fly as well, since the computation time is in the sub-microsecond range. There, we varied the code word width ($|\mathbb{C}| \in \{7, 15, 31, 63, 127\}$) and for each the bit width of the A for which to compute the inverse ($2 \leq |A| \leq |\mathbb{C}|$). Each curve represents an average over $10.000 \cdot (|\mathbb{C}| - 2)$ computations.

Conclusion. The micro benchmarks proof that using the multiplicative inverse makes AN coding well competitive, which is why *Continuous* and *Reencoding* perform so good at SSB.

8 RELATED WORK

A few years ago, Boehm et al. [7] already pointed out the lack of data management techniques dealing with an increasing number of bit flips in main memory as an increasingly relevant source. In [60, 61], the combination of TMR and database systems was investigated. The most recent database-specific work of Kolditz et al. [43] hardens index structures like B-Trees using various techniques for online error detection. They have shown that slightly increasing data redundancy at the right places by incorporating context knowledge increases error detection significantly. *AHEAD* can be extended by their work (cf. Sections 4.1 and 6.1). Furthermore, checksums are usually utilized to check for data integrity. For instance, HDFS computes a checksum of each data block and stores it in separate, hidden files [72]. Whenever a client fetches a data block, it verifies the retrieved data using the associated checksum [72], but this is only done for disk blocks leaving in-memory data vulnerable.

Moreover, AN coding has also been used for software-based fault tolerance [29, 69, 79]. For instance, the work of Schiffel [69] allows to encode existing software binaries or to add encoding at compile time, where not all variables' states need to be known in advance. However, in her work she only describes encoding integers of size $|\mathbb{D}| \in \{1, 8, 16, 32\}$ bits and pointers, where the encoded values are always 64 bits large. Furthermore, protecting processors by AN coding was also suggested in [22].

9 CONCLUSION AND OUTLOOK

Future hardware becomes less reliable in total and scaling up todays hardware-based protection introduces too much overhead [11, 28, 63, 70]. Therefore, a shift towards mitigating these reliability issues at higher layers, rather than only dealing with these issues in hardware was initiated [28, 63, 70]. However, traditional general-purpose software-based protection techniques mainly rely on dual modular redundancy (DMR) to detect errors [25, 57, 65, 63]. For database systems, DMR introduces high overhead, because all data has to be duplicated and every query is executed redundantly including a result comparison as the final step. To overcome these drawbacks, we have presented our novel *adaptable and on-the-fly* ***error detection*** *approach* called *AHEAD*. With our approach, we achieve the following properties: (1) *AHEAD* detects (i) errors (multi-bit flips) that modify data stored in main memory or transmit over an interconnect and (ii) errors induced during computations, (2) *AHEAD* provides configurable error detection capabilities to be able to adapt to different error models at run-time, and (3) *AHEAD* drastically reduces the overhead compared to DMR and errors are continuously detected at query processing. Thus, *AHEAD* is the first comprehensive database-specific approach to tackle the challenge of resilient query processing on unreliable hardware. As next, the following steps have to be done:

Optimization and Extension: In Section 6 we show that our current storage overhead is suboptimal. Bit-level data compression as in [82, 83] could be one solution. While data hardening and lightweight compression [1, 15] are orthogonal to each other, their interplay is very important to keep the overall memory footprint of data as low as possible. With hardening, compression gains even more significance, since it can reduce the newly introduced storage overhead. However, combining both may be challenging and has to be investigated in detail. Furthermore, *AHEAD* cannot detect errors in logic operations, whereas these are frequently used in database systems, e.g., in novel column storage layouts like BitWeaving [49] and ByteSlice [21]. This domain must be considered separately, for this, *AHEAD* can serve as the basis. Further extensions of *AHEAD* could be (1) the use of code word accumulators to do detection every nth code word, trading accuracy against performance, or (2) hardening of database meta data and block or string data.

Error Correction: So far, we were concerned with the continuous detection of bit flips during query processing. Next, continuous error correction should be considered so that detected bit flips are corrected during query processing. With *AHEAD*, we are able detect bit flips on value granularity and can find out where the error occurred. Based on that, specific correction techniques can be developed and integrated in the query processing. For example, if we detect a faulty code word in the inputs of an operator, we can retransmit it, possibly several times, to correct errors induced during transmission. If we get a valid code word, processing can continue with this correct code word. If we get an invalid code word, we can assume that bits are flipped in main memory and then we require an appropriate technique for error correction. For that, correcting errors in the memory requires data redundancy in any case.

Cross-Layer Approach: AHEAD is primarily a software approach. Another interesting research direction would examine the interplay of hardware and software protection mechanisms. In particular it should be scrutinized what should be done in hardware and what should be done in software. From our point of view, *AHEAD* could serve as foundation for such a novel research direction [63].

ACKNOWLEDGMENTS

This work is partly funded by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (Resilience Path). Parts of the evaluation hardware were generously provided by the Dresden GPU Center of Excellence. We also thank the anonymous reviewers for their constructive feedback to improve this paper.

REFERENCES

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. "Integrating compression and execution in column-oriented database systems". In: *SIGMOD*. 2006, pp. 671–682.

[2] Daniel Abadi et al. "The Beckman report on database research". In: *Commun. ACM* 59.2 (2016), pp. 92–99.

[3] Daniel Abadi et al. "The Design and Implementation of Modern Column-Oriented Database Systems". In: *Foundations and Trends in Databases* 5.3 (2013), pp. 197–280.

[4] Algirdas Avizienis. "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design". In: *IEEE Trans. Computers* 20.11 (1971), pp. 1322–1331.

[5] Algirdas Avizienis. "The N-Version Approach to Fault-Tolerant Software". In: *IEEE Trans. Software Eng.* 11.12 (1985), pp. 1491–1501.

[6] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. "Dictionary-based order-preserving string compression for main memory column stores". In: *SIGMOD*. 2009, pp. 283–296.

[7] Matthias Böhm, Wolfgang Lehner, and Christof Fetzer. "Resiliency-Aware Data Management". In: *PVLDB* 4.12 (2011), pp. 1462–1465.

[8] Peter A. Boncz and Martin L. Kersten. "MIL Primitives for Querying a Fragmented World". In: *VLDB J.* 8.2 (1999), pp. 101–119.

[9] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. "Breaking the memory wall in MonetDB". In: *Commun. ACM* 51.12 (2008), pp. 77–85.

[10] Peter Alexander Boncz. "Monet; a next-Generation DBMS Kernel For Query-Intensive Applications". PhD thesis. University of Amsterdam, 2002.

[11] Shekhar Y. Borkar. "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation". In: *IEEE Micro* 25.6 (2005), pp. 10–16.

[12] Shekhar Borkar and Andrew A. Chien. "The future of microprocessors". In: *Commun. ACM* 54.5 (2011), pp. 67–77.

[13] Sebastian Breß, Henning Funke, and Jens Teubner. "Robust Query Processing in Co-Processor-accelerated Databases". In: *SIGMOD*. 2016, pp. 1891–1906.

[14] George P. Copeland and Setrag Khoshafian. "A Decomposition Storage Model". In: *SIGMOD*. 1985, pp. 268–279.

[15] Patrick Damme et al. "Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses)". In: *EDBT*. 2017, pp. 72–83.

[16] Timothy J Dell. "A white paper on the benefits of chipkill-correct ECC for PC server main memory". In: *IBM Microelectronics Division* 11 (1997).

[17] Cristian Diaconu et al. "Hekaton: SQL server's memory-optimized OLTP engine". In: *SIGMOD*. 2013, pp. 1243–1254.

[18] Jaeyoung Do et al. "Query processing on smart SSDs: opportunities and challenges". In: *SIGMOD*. 2013, pp. 1221–1230.

[19] Dan Ernst et al. "Razor: circuit-level correction of timing errors for low-power operation". In: *IEEE Micro* 24.6 (2004), pp. 10–20.

[20] Hadi Esmaeilzadeh et al. "Dark Silicon and the End of Multicore Scaling". In: *IEEE Micro* 32.3 (2012), pp. 122–134.

[21] Ziqiang Feng et al. "ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout". In: *SIGMOD*. 2015, pp. 31–46.

[22] P Forin. "Vital Coded Microprocessor: Principles and Application for Various Transit Systems". In: *IFAC-GCCT* (1989).

[23] Free Software Foundation. *The GNU Multiple Precision Arithmetic Library*. https://gmplib.org/. Nov. 2016.

[24] Brian Gladman et al. *MPIR: Multiple Precision Integers and Rationals*. http://mpir.org/. Nov. 2016.

[25] Olga Goloubeva et al. *Software-implemented hardware fault tolerance*. Springer Science & Business Media, 2006.

[26] Richard W Hamming. "Error detecting and error correcting codes". In: *Bell System technical journal* 29.2 (1950).

[27] Jörg Henkel. "Emerging Memory Technologies". In: *IEEE Design & Test* 34.3 (2017), pp. 4–5.

[28] Jörg Henkel et al. "Reliable on-chip systems in the nano-era: lessons learnt and future trends". In: *DAC*. 2013, 99:1–99:10.

[29] Martin Hoffmann et al. "A Practitioner's Guide to Software-Based Soft-Error Mitigation Using AN-Codes". In: *HASE*. 2014, pp. 33–40.

[30] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design". In: *ASPLOS*. 2012, pp. 111–122.

[31] Eishi Ibe et al. "Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule". In: *IEEE Transactions on Electron Devices* 57.7 (2010), pp. 1527–1538.

[32] Stratos Idreos et al. "MonetDB: Two Decades of Research in Column-oriented Database Architectures". In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 40–45.

[33] K Itoh et al. "A single 5V 64K dynamic RAM". In: *ISSCC*. Vol. 23. 1980, pp. 228–229.

[34] Lei Jiang, Youtao Zhang, and Jun Yang. "Mitigating Write Disturbance in Super-Dense Phase Change Memories". In: *DSN*. 2014, pp. 216–227.

[35] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. "Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources". In: *PVLDB* 10.7 (2017), pp. 733–744.

[36] D. Kaur and D. Wedding. "Reliability of Hamming code transmission versus error probability on message bits". In: *Microelectronics Reliability* 34.7 (1994).

[37] Samira Manabi Khan, Donghyuk Lee, and Onur Mutlu. "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM". In: *DSN*. 2016, pp. 239–250.

[38] Samira Khan et al. "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study". In: *SIGMETRICS Perform. Eval. Rev.* 42.1 (June 2014), pp. 519–532.

[39] Jangwoo Kim et al. "Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding". In: *Symposium on Microarchitecture*. 2007, pp. 197–209.

[40] Yoongu Kim et al. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *ISCA*. 2014, pp. 361–372.

[41] Thomas Kissinger et al. "QPPT: Query Processing on Prefix Trees". In: *CIDR*. 2013.

[42] Masanobu Kohara et al. "Mechanism of electromigration in ceramic packages induced by chip-coating polyimide". In: *IEEE Transactions on Components, Hybrids, and Manufacturing Technology* 13.4 (1990), pp. 873–878.

[43] Till Kolditz et al. "Online bit flip detection for in-memory B-trees on unreliable hardware". In: *DaMoN*. 2014, 5:1–5:9.

[44] Emre Kultursay et al. "Evaluating STT-RAM as an energy-efficient main memory alternative". In: *ISPASS*. 2013, pp. 256–267.

[45] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. "Oracle TimesTen: An In-Memory Database for Enterprise Applications". In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 6–13.

[46] Benjamin C. Lee et al. "Architecting phase change memory as a scalable dram alternative". In: *ISCA*. 2009, pp. 2–13.

[47] Christiane Lemieux. *Monte Carlo and Quasi-Monte Carlo Sampling*. Springer, 2009. ISBN: 978-1441926760.

[48] Feng Li et al. "Accelerating Relational Databases by Leveraging Remote Memory and RDMA". In: *SIGMOD*. 2016, pp. 355–370.

[49] Yinan Li and Jignesh M. Patel. "BitWeaving: Fast Scans for Main Memory Data Processing". In: *SIGMOD*. 2013, pp. 289–300.

[50] Jamie Liu et al. "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms". In: *SIGARCH Comput. Archit. News* 41.3 (June 2013), pp. 60–71.

[51] Sparsh Mittal. "A Survey of Soft-Error Mitigation Techniques for Non-Volatile Memories". In: *Computers* 6.1 (2017), p. 8.

[52] Todd K Moon. "Error correction coding". In: *Mathematical Methods and Algorithms. Jhon Wiley and Son* (2005).

[53] Wojciech Mula, Nathan Kurz, and Daniel Lemire. "Faster Population Counts using AVX2 Instructions". In: *CoRR* (2016).

[54] Onur Mutlu. "The RowHammer problem and other issues we may face as memory becomes denser". In: *DATE*. 2017, pp. 1116–1121.

[55] Thomas Neumann. *The price of correctness*. http://databasearchitects.blogspot.de/2015/12/the-price-of-correctness.html. Nov. 2016.

[56] Patrick O'Neil et al. "The Star Schema Benchmark and Augmented Fact Table Indexing". In: *TPCTC 2009: Performance Evaluation and Benchmarking*. Berlin, Heidelberg: Springer, 2009, pp. 237–252. DOI: 10.1007/978-3-642-10424-4_17.

[57] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. "Error detection by duplicated instructions in super-scalar processors". In: *IEEE Transactions on Reliability* 51.1 (2002), pp. 63–75.

[58] Ismail Oukid et al. "FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory". In: *SIGMOD*. 2016, pp. 371–386.

[59] William Wesley Peterson and Daniel T Brown. "Cyclic codes for error detection". In: *IRE* 49.1 (1961), pp. 228–235.

[60] Frank M. Pittelli and Hector Garcia-Molina. "Database Processing with Triple Modular Redundancy". In: *SRDS*. 1986, pp. 95–103.

[61] Frank M. Pittelli and Hector Garcia-Molina. "Reliable Scheduling in a TMR Database System". In: *ACM Trans. Comput. Syst.* 7.1 (1989), pp. 25–60.

[62] Fred J. Pollack. "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies". In: *Symposium on Microarchitecture*. 1999, p. 2.

[63] Semeen Rehman, Muhammad Shafique, and Jörg Henkel. *Reliable Software for Unreliable Hardware - A Cross Layer Perspective*. Springer, 2016.

[64] Steven K. Reinhardt and Shubhendu S. Mukherjee. "Transient fault detection via simultaneous multithreading". In: *ISCA*. 2000, pp. 25–36.

[65] George A. Reis et al. "SWIFT: Software Implemented Fault Tolerance". In: *CGO*. 2005, pp. 243–254.

[66] Michael C. Ring. *MAPM, A Portable Arbitrary Precision Math Library in C*. http://www.tc.umn.edu/~ringx004/mapm-main.html. Nov. 2016.

[67] Ronald Linn Rivest. *The MD5 Message-Digest Algorithm*. Nov. 2016. URL: https://tools.ietf.org/html/rfc1321.

[68] Jimi Sanchez. "A Review of Star Schema Benchmark". In: *CoRR* abs/1606.00295 (2016).

[69] Ute Schiffel. "Hardware error detection using AN-Codes". PhD thesis. Dresden University of Technology, 2011.

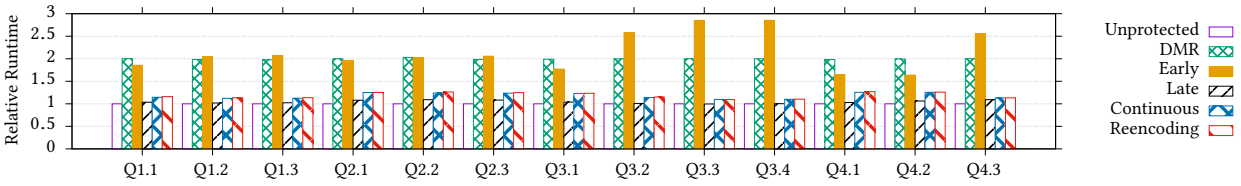


Figure 11: Relative SSB runtimes for scalar execution (average over all scale factors).

[70] Muhammad Shafique et al. "Multi-layer software reliability for unreliable hardware". In: *it - Information Technology* 57.3 (2015), pp. 170–180.

[71] Erez Shmueli et al. "Database encryption: an overview of contemporary challenges and design considerations". In: *SIGMOD Record* 38.3 (2009), pp. 29–34.

[72] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *MSST*. 2010, pp. 1–10.

[73] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. "Ensuring Data Integrity in Storage: Techniques and Applications". In: *StorageSS*. 2005.

[74] Michael Spica and T. M. Mak. "Do We Need Anything More Than Single Bit Error Correction (ECC)?" In: *MTDT*. 2004, pp. 111–116.

[75] Michael Stonebraker et al. "C-Store: A Column-oriented DBMS". In: *VLDB*. 2005, pp. 553–564.

[76] Stephen Y. H. Su and Edgar DuCasse. "A hardware redundancy reconfiguration scheme for tolerating multiple module failures". In: *IEEE Transactions on Computers* 3.C-29 (1980), pp. 254–258.

[77] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. "Simultaneous Multithreading: Maximizing On-Chip Parallelism". In: *ISCA*. 1995, pp. 392–403.

[78] Peter Ulbrich, Martin Hoffmann, and Christian Dietrich. *CoRed: Experimental Results*. https://www4.cs.fau.de/Research/CoRed/experiments/. July 2017.

[79] Peter Ulbrich et al. "Eliminating single points of failure in software-based redundancy". In: *EDCC*. 2012, pp. 49–60.

[80] Henry S Warren. *Hacker's delight*. Pearson Education, 2013.

[81] Matthias Werner et al. "Multi-GPU Approximation for Silent Data Corruption of AN Codes". In: *Further Improvements in the Boolean Domain*. Ed. by Bernd Steinbach. Cambridge Scholars Publishing, 2018. Chap. 2.3, pp. 136–155.

[82] Thomas Willhalm et al. "SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units". In: *Proc. VLDB Endow.* (2009).

[83] Thomas Willhalm et al. "Vectorizing database column scans with complex predicates". In: *ADMS*. 2013, pp. 1–12.

[84] Ian H. Witten, Radford M. Neal, and John G. Cleary. "Arithmetic Coding for Data Compression". In: *Commun. ACM* 30.6 (1987), pp. 520–540.

[85] J. Wolf, A. Michelson, and A. Levesque. "On the Probability of Undetected Error for Linear Block Codes". In: *IEEE Transactions on Communications* 30.2 (1982).

[86] H.-S. Philip Wong et al. "Metal-Oxide RRAM". In: *Proceedings of the IEEE* 100.6 (2012), pp. 1951–1970.

[87] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. "Vectorwise: A Vectorized Analytical DBMS". In: *ICDE*. 2012, pp. 1349–1350.

[88] Marcin Zukowski et al. "Super-Scalar RAM-CPU Cache Compression". In: *ICDE*. 2006, p. 59.

A SSB SCALAR RUNTIME PERFORMANCE

Figure 11 shows the relative scalar runtimes for all 13 SSB queries as averages over all 10 scale factors and 10 runs each. The **Unprotected** variant is the baseline. **DMR** shows the expected 100% runtime overhead, whereas **Early** has very high overheads between 64% and 185%, where the main overhead results from the slow Δ operator which decodes all encoded columns. The overhead of **Late Detection** is always below 10%. **Continuous** and **Reencoding** runtimes show overheads between 10% and 27%.

B PROTECTION TECHNIQUES

Hardware-based protection can be done on three layers [63]: (i) transistor, (ii) circuit, and (iii) architectural. On the *transistor layer*, several techniques have been proposed to harden transistors against radiation events like alpha particles or neutron strikes [33, 42]. For example, thick polyamide can be used for alpha particle protection [33, 42]. However, this technique cannot be utilized for neutron strikes [63]. In general, techniques at this layer have in common that the protection results in adopted fabrication processes using specialized materials [33, 42, 63]. Therefore, these techniques are very effective, but they produce (i) substantial overhead in terms of area and cost, and (ii) immense validation and verification costs.

At the *circuit layer*, redundant circuits and error detection/correction circuits are prominent examples [16, 19, 39, 63]. For instance, the RAZOR approach introduces shadow flip flops in the pipeline to recover from errors in logic gates [19]. Memories and caches are usually protected using error correcting codes (ECC) or parity techniques. Current ECC memories are based on Hamming using a (72,64) code, meaning that 64 bits of data are enhanced with 8 bits of parity allowing single error correction and double error detection. However, this is not sufficient to address multi-bit flips. To tackle multi-bit flips advanced ECC schemes have to be used. Examples are (i) IBM's Chipkill approach, wich computes the parity bits from different memory words and even separate DIMMs instead of physically adjacent bits [16], and (ii) [39], which shows that other ECC codes like BCH-codes [52] can be realized in hardware to be able to correct e.g., 8-bit flips and detect 9-bit flips for 64 bits of data. However, this increases the number of transistors in hardware and consequently impacts the energy demand, the overhead growing quickly as the code strength is increased [39]. Additionally, reading and computing the enhanced ECC bits can be a performance bottleneck during read operations [39]. To mitigate *disturbance errors* at this layer, hardware vendors improve inter-cell isolation, but this is challenging due to ever-decreasing feature sizes and higher densities [40, 54].

At the *architectural layer*, the protection is based upon the redundant execution either in space (using duplicated hardware units) or in time (using the same hardware multiple times for redundant execution and comparing the results). Dual Modular Redundancy (DMR) and Triple Modular Redundancy (TMR) are traditional approaches. Generally, these techniques lead to an increased power usage which may potentially increase the temperature [63]. Increased temperatures lead to higher soft error rate and increased aging [63]. To lower these effects, multi-/manycore architectures provide soft error tolerance through the availability of a high number of cores. Idle cores can now be exploited to provide redundancy either at the hardware level (using redundant instructions or redundant threads) or operating system level (using redundant thread processes). For example, the Simultaneous Redundant Threading (SRT) approach [64] adapts the concept of Simultaneous Multithreading (SMT) [77]. SMT was proposed to improve performance via executing program codes of different applications in a simultaneous multithreaded fashion on multiple functional units inside a given processor. In contrast to that, SRT executes two redundant threads of the same application on multiple functional units and then performs the output comparison.

	exact			$\sigma_{\text{grid,1D,4·GPU}}$		
k	t_{CPU}	$t_{1\cdot\text{GPU}}$	$t_{4\cdot\text{GPU}}$	t_M	Δ_M	M
8	7 ms	1 ms	3 ms	6 ms	0.0232	101
16	376 ms	130 ms	41 ms	11 ms	0.0031	1001
24	382 min	99 min	27 min	354 ms	0.0053	1001
32	–	–	–	5 min	–	1001

Table 2: Computing the distance distributions of AN codes for $A = 61$. Average values after 5 runs. CPU: 2×E5-2680 v3 Haswell 12-core 2.50 GHz**, gcc5.3, OpenMP 4.0. GPU: NVIDIA Tesla K80, CUDA 7.5**

To summarize, hardware-based protection techniques are usually very effective, but they also have major drawbacks in terms of (i) high area overhead leading at the same time to more power overhead and (ii) performance penalties. Furthermore, the high verification/validation costs make the reliable hardware design and development very expensive and time consuming [63]. To overcome these non-negligible drawbacks, a rich set of software-based techniques has evolved.

Classical software-based protection techniques are [25, 63]: (i) N-version programming, (ii) code redundancy, (iii) control flow checking, and (iv) checkpoint recovery. For instance, N-version programming [5] is based on implementing multiple program versions of the same specification which reduces the probability of identical errors occurring in two or more versions. State-of-the-art redundancy-based techniques are Error Detection using Duplicated Instructions (EDDI) [57] and Software Implemented Fault Tolerance (SWIFT) [65]. Both provide software reliability by duplicating instructions, and inserting comparison and checking instructions. However, these techniques incur significant performance overheads [57, 65].

C COMPUTING THE SDC PROBABILITY

There is plenty of work on evaluating the probability of undetected errors for linear block codes[13] and Hamming code in particular [36, 85, 52]. For AN codes, this has only been done for 8- and 16-bit data and As up to 8 and 16 bits [29, 78]. However, this is insufficient for the database domain, because (1) possibly all data bit widths between 1 and 64 bits are to be supported [83], and (2) larger As may be required for future error models. To overcome that, we developed a practical methodology for determining the probability of SDC for non-systematic, non-linear[14] coding schemes in general and independent of a specific error model [81]. For this, we use the following definitions: A code $\mathbb{C}$ (EDC/ECC) is defined by the triplet $(n, k, d_{\min})$, were $n = |\mathbb{C}|$ is the code word width, k is the data width, and $d_{\min}$ is the minimum Hamming distance between any of the code's valid code words. Further, b denotes the number of flipped bits. For AN coding, the definitions from Section 3 further apply.

Conceptually, we model the space of all *valid* code words as an undirected, fully connected, weighted graph, where the code words are the vertices. Each valid code word is connected with every other valid one and the edge weights denote the *Hamming distance* d_H between the two code words. Thus, we only concentrate on the *transitions* originating from valid code words to other valid ones, *regardless of the coding*. Then, computing SDC probabilities requires two steps. First, we build a histogram over all transition weights, by which we get the numbers c_b of undetectable b-bit flips:

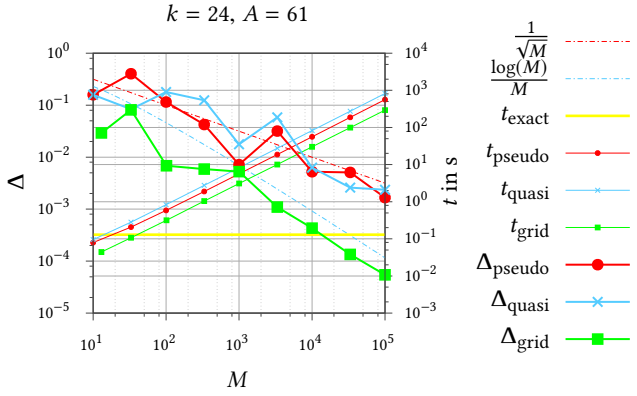


Figure 12: Convergence of maximum relative error Δ and run-time t according to the number of iterations M.

Definition 1. *For a given code, c_b denotes the number of transitions of weight b between valid code words.*

The set $\{c_b | b \in \{1, \ldots, k\}\}$ is called *weight distribution*. For AN coding, c_b^A represents the count for a given A. Second, we relate each c_b to the respective total number of possible b-bit flips $\binom{n}{b}$. Consequently, there can be $2^k \cdot \binom{n}{b}$ b-bit flips in all valid code words. This number includes all transitions from any *valid* codeword to any other *possible* code word. By that, when computing the weight distribution, we also have to count the transitions modeled in the graph twice (i.e., for each direction), because with a single error pattern we can make a transition in both direction. In total, this results in the SDC probability for b-bit flips, which we denote as:

$$p_b = \frac{c_b}{2^k \cdot \binom{n}{b}} \tag{14}$$

Now, the challenge is to obtain c_b in an efficient manner. For non-linear codes like AN coding, c_b must be *counted* in a brute force manner to the best of our knowledge. For AN coding, the convolution of the multiplication cannot be described in a way which allows simplifications. We use the following function to describe the naive approach:

$$\delta_b(x, y) = \begin{cases} 1, & \text{if } d_H(x, y) = b, \\ 0, & \text{if } d_H(x, y) \neq b, \end{cases} \qquad 0 \leq b \leq n\,.$$

Then, we can compute the distance distribution as

$$c_b = \sum_{\alpha \in \mathbb{C}} \sum_{\beta \in \mathbb{C}} \delta(\alpha, \beta)\,. \tag{15}$$

The complexity of Equation (15) is $O(4^k)$, i.e. with each additional data bit, there are four times as many distances. In practice, it is half as much due to the symmetry (only one edge is computed and then counted twice). For parameter optimization, this might be run many thousands of times. Especially, for AN coding each odd A must be examined again for each data width $|\mathbb{D}_\Theta|$ and with

[13] For linear codes, the linear combination (*exclusive* OR) of two valid code words is *always* also a valid code word.

[14] In non-linear codes, the linearity property is not always satisfied.

Algorithm 2 AN code distance distribution – basic algorithm

Input: $k \geq 2$
Input: Value $A > 0$, $n = k + h$, $h = \lceil \log_2(A) \rceil$
Input: Initial distance distribution $c_b^A = 0,\ b = 0, \ldots, n$
Output: Distance distribution c^A of code C_A

1: **for** $\alpha = 0, \ldots, (2^k - 1)$ **do** ▷ outer loop is parallelized on GPU[s]
2: **for** $\beta = \sigma_{\text{grid}}(r),\ r = 0, \ldots, M$ **do** ▷ inner loop is processed by each thread
3: $b \leftarrow d_H(A\alpha, A\beta)$
4: $c_b^A \leftarrow c_b^A + 1$
5: **end for**
6: **end for**
7: **return** c^A

each additional bit, the number of candidates doubles. We call this naive approach *exact*, as it examines all code words. For AN coding, Table 2 shows runtimes for the exact computation of the weight distribution for a single A using a single CPU, a single GPU, or a small cluster of 4 GPUs.

To mitigate the complexity, we use a sampling-based approach, which approximates the weight distribution by comparing only a subset of all code words. Here, the main problem is the *distribution* function for choosing the subset of code words. We investigated three different distributions: pseudo-random (σ_{pseudo}), quasi-random (σ_{quasi}), and grid-point (σ_{grid}). Note that σ_{pseudo} is prone to clustering, while σ_{quasi} fills the space more uniformly. The probabilistic error of Monte-Carlo (pseudo-random) is known to be $O(1/\sqrt{M})$ and for quasi-Monte-Carlo it is $O((\log M)^q/M)$ with number of dimensions q and number of iterations M [47]. The grid-point approach chooses regularly aligned samples, given by $\sigma_{\text{grid}}(r) = {}^{(2^k \cdot r)}/_M$. If $M = 2^k$, then the grid sampling yields the correct result, while random numbers still miss the solution due to collisions and gaps. Figure 12 shows a comparison between the three distributions of convergence and runtime for the case $k = 24$ and $A = 61 \Rightarrow n = 30$ and includes the theoretic Monte-Carlo error boundaries. Pseudo- and quasi-random numbers were generated with the cuRAND library. The 1D grid approximation outperforms the random distributions in virtually all cases, yielding smaller error Δ and lower runtime t. It is, furthermore, directly influenced by the value of M, and we found that *odd* values lead to much smaller errors than even ones.

Algorithm 2 shows the 1D grid approach for enumerating the weight distribution of an AN code. For GPU clusters, we distribute the outer loop evenly across the GPUs. When symmetry is exploited in line 2, the workload size of each GPU is computed by:

$$\lceil 2^k \omega_{i+1} \rceil - \lceil 2^k \omega_i \rceil,\ w_i = 1 - \sqrt{1 - i/N},\ 0 \leq i < N_{=\#\text{GPUs}} \quad (16)$$

ω_i is the solution of $\int_i^{i+1} 1 - x \, \mathrm{d}x = 1/N$ for equal work size areas. The maximal relative error of the estimation $\hat{c}_b^A$ is given by $\Delta = \max_{b>0} \frac{|c_b^A - \hat{c}_b^A|}{c_b^A}$ ($b = 0$ is omitted due to $c_0^A = 2^k$).

Algorithm 2 can be parallelized on GPUs, since the Hamming distances of two code words can be computed independently. We use CUDA C/C++ for programming Nvidia GPUs. As registers of GPUs are 32-bit wide, the multi-GPU implementation uses 32-bit integers as long as the array elements in a thread do not overflow. From Equation 14 follows $\max_b c_b^A \leq \max_b 2^k \binom{n}{b} = 2^k \binom{n}{n/2}$ and the upper bound for using 32-bit integers is:

$$c_{b,\text{thread}}^A \leq \frac{2^k \binom{n}{n/2}}{\text{threads}} < 2^{32} .$$

The GPU uses 64 bits for the global array, so the highest data bit width for the GPU algorithm is $k = 33$. The range for each GPU is between $\lceil 2^k \omega_i \rceil$ and $\lceil 2^k \omega_{i+1} \rceil$ from Equation 16. We use thread-local arrays and one global array to avoid memory contention. Since local array indexing is dynamic and non-uniform, it cannot be stored into the fast registers, as they are not addressable at run-time. Hence, the local array is stored in local memory, which is L1 cached thread-private global memory. To get scalable and flexible kernels, the outer loop strides by the size of a CUDA grid (threads per block × blocks per grid). The kernel is called with `blocks=32` · numberOfMultiprocessors and 128 threads per block. After the local histogram is filled, atomic operations are used to add the values to the global distance distribution. The respective runtimes are shown in Table 2 on the right half. More details can be found in [81].

$\lvert\mathbb{D}_\Theta\rvert$	minimal detectable bit flip weight						
	1	2	3	4	5	6	7
1	**3**/2	**7**/3	15/4	**31**/5	63/6	**127**/7	255/8
2	**3**/2	**13**/4	**53**/6	213/8	**853**/10	3285/12	13141/14
3	**3**/2	**29**/5	45/6	**467**/9	1837/11	**7349**/13	23733/15
4	**3**/2	27/5	**89**/7	933/10	6777/13	31385/15	
5	**3**/2	**29**/5	117/7	933/10	7085/13	31373/15	
6	**3**/2	**29**/5	**233**/8	1899/11	7837/13	62739/16	
7	**3**/2	**29**/5	217/8	1803/11	**13963**/14	55831/16	
8	**3**/2	**29**/5	**233**/8	1939/11	**13963**/14	55831/16	
9	**3**/2	**29**/5	185/8	1939/11	15717/14	55831/16	
10	**3**/2	**61**/6	185/8	**3739**/12	27425/15		
11	**3**/2	**61**/6	451/9	**3739**/12	27425/15		
12	**3**/2	**61**/6	**463**/9	3737/12	29925/15		
13	**3**/2	**61**/6	**463**/9	3349/12	27825/15		
14	**3**/2	**61**/6	**463**/9	6717/13	63877/16		
15	**3**/2	**61**/6	**463**/9	7785/13	63877/16		
16	**3**/2	**61**/6	**463**/9	7785/13	63877/16		
17	**3**/2	**61**/6	393/9	7785/13	63859/16		
18	**3**/2	**61**/6	**947**/10	7785/13	63859/16		
...	...						
28	**3**/2	111/7	951/10	29685/15			
29	**3**/2	111/7	835/10	*29685/15			
30	**3**/2	125/7	835/10	*31693/15			
31	**3**/2	125/7	**881**/10	*32211/15			
32	**3**/2	125/7	**881**/10	*32417/15			

Table 3: Smallest super *A*s per minimal detectable bit flip weight in the form $A/|A|$. Bold numbers are prime. * obtained through grid approximation.

D GITHUB

Finally, we would like to point out our project *BRICS-DB* on GitHub[15] where our prototypical *AHEAD* implementation is available, as well as our GPU implementation for computing distance distributions (SDC probability). We also provide the full table for all smallest super *A*s per bit flip weight which we computed until now. More super *A*s will be calculated and all updated information will be provided on github. Table 3 shows an excerpt of the current table of super *A*s.

[15] https://brics-db.github.io/