HARP: Practically and Effectively Identifying Uncorrectable Errors in Memory Chips That Use On-Die Error-Correcting Codes

Minesh Patel
ETH Zürich

Geraldo F. Oliveira
ETH Zürich

Onur Mutlu
ETH Zürich

ABSTRACT

Aggressive storage density scaling in modern main memories causes increasing error rates that are addressed using error-mitigation techniques. State-of-the-art techniques for addressing high error rates identify and repair bits that are at risk of error from within the memory controller. Unfortunately, modern main memory chips internally use on-die error correcting codes (on-die ECC) that obfuscate the memory controller's view of errors, complicating the process of identifying at-risk bits (i.e., error profiling).

To understand the problems that on-die ECC causes for error profiling, we analytically study how on-die ECC changes the way that memory errors appear outside of the memory chip (e.g., to the memory controller). We show that on-die ECC introduces statistical dependence between errors in different bit positions, raising three key challenges for practical and effective error profiling: on-die ECC (1) exponentially increases the number of at-risk bits the profiler must identify; (2) makes individual at-risk bits more difficult to identify; and (3) interferes with commonly-used memory data patterns that are designed to make at-risk bits easier to identify.

To address the three challenges, we introduce Hybrid Active-Reactive Profiling (HARP), a new error profiling algorithm that rapidly achieves full coverage of at-risk bits based on two key insights. First, errors that on-die ECC fails to correct have two sources: (1) direct errors from raw bit errors in the data portion of the ECC word and (2) indirect errors that on-die ECC introduces when facing uncorrectable errors. Second, the maximum number of indirect errors that can occur concurrently is limited to the correction capability of on-die ECC. HARP's key idea is to first identify all bits at risk of direct errors using existing profiling techniques with the help of small modifications to the on-die ECC mechanism. Then, a secondary ECC within the memory controller with correction capability equal to or greater than that of on-die ECC can safely identify bits at-risk of indirect errors, if and when they fail.

We evaluate HARP in simulation relative to two state-of-the-art baseline error profiling algorithms. We show that HARP achieves full coverage of all at-risk bits faster (e.g., 99th-percentile coverage 20.6%/36.4%/52.9%/62.1% faster, on average, given 2/3/4/5 raw bit errors per ECC word) than the baseline algorithms, which sometimes fail to achieve full coverage. We perform a case study of how each profiler impacts the system's overall bit error rate (BER) when using a repair mechanism to tolerate DRAM data-retention errors. We show that HARP identifies all errors faster than the best-performing baseline algorithm (e.g., by 3.7× for a raw per-bit error probability of 0.75). We conclude that HARP effectively overcomes the three error profiling challenges introduced by on-die ECC.

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480061

CCS CONCEPTS

• **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Hardware** → **Memory test and repair**.

KEYWORDS

On-Die ECC, DRAM, Memory Test, Repair, Error Profiling, Error Modeling, Memory Scaling, Reliability, Fault Tolerance

ACM Reference Format:
Minesh Patel, Geraldo F. Oliveira, and Onur Mutlu. 2021. HARP: Practically and Effectively Identifying Uncorrectable Errors in Memory Chips That Use On-Die Error-Correcting Codes. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3466752.3480061

1 INTRODUCTION

Modern memory technologies that are suitable for main memory (e.g., Dynamic Random Access Memory (DRAM) [38, 116], Phase-Change Memory (PCM) [17, 101, 102, 151, 177], Spin-Transfer Torque RAM (STT-RAM) [56, 97]) all suffer from various error mechanisms that play a key role in determining reliability, manufacturing yield, and operating characteristics such as performance and energy efficiency [17, 21, 45, 62, 74, 87, 97, 101, 102, 107, 116, 144, 174, 184]. Unfortunately, as memory designers shrink (i.e., scale) memory process technology node sizes to meet ambitious capacity, cost, performance, and energy efficiency targets, worsening reliability becomes an increasingly significant challenge to surmount [10, 21, 54, 74, 87, 89, 91, 107, 116, 129, 132, 136, 144, 150, 170]. For example, DRAM process technology scaling exacerbates cell-to-cell variation and noise margins, severely impacting error mechanisms that constrain yield, including cell data-retention [21, 46, 47, 54, 74, 76, 77, 93, 110, 136, 144, 147, 161, 171] and read-disturb [40, 52, 87, 91, 131, 132, 141] phenomena. Similarly, emerging main memory technologies suffer from various error mechanisms that can lead to high error rates if left unchecked, such as limited endurance, resistance drift, and write disturbance in PCM [10, 62, 73, 88, 101, 105] and data retention, endurance, and read disturbance in STT-RAM [9, 25, 28, 134, 152, 170]. Therefore, enabling reliable system operation in the presence of scaling-related memory errors is a critical research challenge for allowing continued main memory scaling.

Error mitigation mechanisms and on-die ECC. Modern systems tolerate errors using *error-mitigation mechanisms*, which prevent errors that occur within the memory chip from manifesting as software-visible bit flips. Different error-mitigation mechanisms target different types of errors, ranging from fine- to coarse-grained mitigation using hardware and/or software techniques. §2.1 reviews main memory error-mitigation mechanisms.

To address increasing memory error rates, memory chip manufacturers have started to incorporate *on-die error-correcting codes* (on-die ECC).[1] On-die ECC is already prevalent among modern DRAM (e.g., LPDDR4 [98, 99, 121, 140, 145, 146], DDR5 [67]) and STT-RAM [39] chips because it enables memory manufacturers to aggressively scale their technologies while maintaining the appearance of a reliable memory chip. Unfortunately, on-die ECC changes how memory errors appear outside the memory chip (e.g., to the memory controller or the system software). This introduces new challenges for designing additional error-mitigation mechanisms at the system level [21, 32, 43, 69, 115, 137, 142, 162] or test a memory chip's reliability characteristics [41, 44, 145, 146].

In this work, we focus on enabling a class of state-of-the-art hardware-based error-mitigation mechanisms known as *repair mechanisms*[2] [61, 92, 93, 109, 111, 112, 135, 136, 150, 157, 158, 162, 168, 171, 176, 179, 180] when used alongside memory chips with on-die ECC. These repair mechanisms operate from outside the memory chip (e.g., from the memory controller) to identify and repair memory locations that are at risk of error (i.e., are known or predicted to experience errors). In particular, prior works [92, 136, 157, 158, 168] show that *bit-granularity* repair mechanisms efficiently tackle high error rates (e.g., $> 10^{-4}$) resulting from aggressive technology scaling by focusing error-mitigation resources on bits that are known to be susceptible to errors.

Fig. 1 illustrates a system that uses both a repair mechanism (within the memory controller) and on-die ECC (within the memory chip). On-die ECC encodes all data provided by the CPU before writing it to the memory. On a read operation, on-die ECC first decodes the stored data, correcting any correctable errors. The repair mechanism then repairs the data before returning it to the CPU. The repair mechanism performs repair operations using a list of bits known to be at risk of error, called an *error profile*.

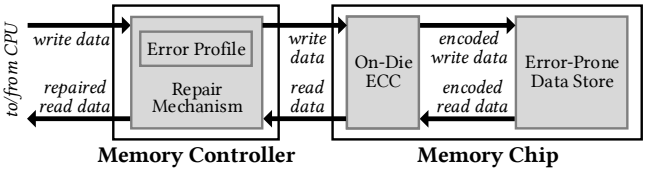


Figure 1: High-level block diagram of a system that uses a repair mechanism with a memory chip that uses on-die ECC.

Error profiling. Repair mechanisms depend on having a practical algorithm for identifying at-risk memory locations to repair. We refer to this as an *error profiling algorithm*. We classify a profiling algorithm as either *active* or *reactive* depending on whether it takes action to search for at-risk memory locations or passively monitors memory to identify errors as they occur during normal system operation. Prior works [11, 12, 24, 27, 29, 30, 40, 48, 76–79, 82, 84, 86, 87, 103, 104, 109–111, 132, 145, 147, 149, 150, 159, 168, 171, 183] propose a variety of algorithms for *active* profiling. In general, these algorithms all search for errors using multiple *rounds* of tests that each attempt to induce and identify errors (e.g., by taking exclusive control of the memory chip [110, 145, 147, 168]). The algorithms maximize the chance of observing errors to identify as many at-risk bits as possible by testing under *worst-case conditions* (e.g., data and access patterns, operating conditions). Only BEEP [145] accounts for the effects of on-die ECC by first reverse-engineering the on-die ECC implementation, so we refer to all other active profiling algorithms as *Naive* in the context of this work. In contrast, proposals for *reactive* profiling passively monitor an error-detection mechanism (typically an ECC) to identify errors as they occur during normal system operation [14, 63, 119, 128, 149, 150, 156, 159].

Regardless of the profiling algorithm, any at-risk bits that the profiler misses will not be repaired by the repair mechanism that the profiler supports. Therefore, achieving practical and effective repair requires a profiling algorithm that quickly and efficiently achieves high coverage of at-risk memory locations.

On-die ECC's impact on error profiling. Unfortunately, on-die ECC *fundamentally changes* how memory errors appear outside of the memory chip: instead of errors that follow well-understood semiconductor error models [66], the system observes obfuscated error patterns that vary with the particular on-die ECC implementation [145, 146]. This is a serious challenge for existing profiling algorithms because, as §4 shows, on-die ECC both (1) *increases* the number of at-risk bits that need to be identified and (2) makes those bits harder to identify. Even a profiler that knows the on-die ECC implementation (e.g., BEEP [145]) cannot easily identify all at-risk bits because it lacks visibility into the error-correction process.

Our goal is to study and address the challenges that on-die ECC introduces for bit-granularity error profiling. To this end, we perform the first analytical study of how on-die ECC affects error profiling. We find that on-die ECC introduces statistical dependence between errors in different bit positions such that, even if raw bit errors (i.e., *pre-correction* errors) occur independently, errors observed by the system (i.e., *post-correction* errors) do not. This raises three new challenges for practical and effective bit-granularity error profiling (discussed in detail in §4).

First, on-die ECC transforms a small set of bits at risk of pre-correction errors into a *combinatorially larger* set of bits at risk of post-correction errors. §4.1 shows how this exponentially increases the number of bits the profiler must identify. Second, on-die ECC ties post-correction errors to specific combinations of pre-correction errors. Only when those specific pre-correction error combinations occur, can the profiler identify the corresponding bits at risk of post-correction errors. §4.2 shows how this significantly slows down profiling. Third, on-die ECC interferes with commonly-used memory data patterns that profilers use to maximize the chance of observing errors. This is because post-correction errors appear only when *multiple* pre-correction errors occur concurrently, which the data patterns must account for. §4.3 discusses the difficulty of defining new data patterns for use with on-die ECC.

[1] Our work applies to *any* memory chip that is packaged with a proprietary ECC mechanism; on-die ECC is one embodiment of such a chip.
[2] We discuss repair mechanisms in detail in §2.2.

To address these three challenges, we introduce Hybrid Active-Reactive Profiling (HARP), a new bit-granularity error profiling algorithm that operates within the memory controller to support a repair mechanism for memory chips with on-die ECC. HARP is based on two key insights. First, given that on-die ECC uses systematic encoding (i.e., data bits are preserved one-to-one during ECC encoding[3]), there are only two possible types of post-correction errors: (1) *direct* errors, corresponding to pre-correction errors within the systematically encoded data bits; and (2) *indirect* errors, resulting from *mistaken* correction operations (i.e., *miscorrections*) on-die ECC performs for *uncorrectable* errors. Second, because on-die ECC corrects a fixed number N of errors, *at most* N indirect errors can occur concurrently (e.g., $N = 1$ for a Hamming code [49]).

Based on these insights, the key idea of HARP is to reduce the problem of profiling a chip *with* on-die ECC into that of a chip *without* on-die ECC by separately identifying direct and indirect errors. HARP consists of two phases. First, an active profiling phase that uses existing profiling techniques to identify bits at risk of direct errors with the help of a simple modification to the on-die ECC read operation that allows the memory controller to read raw data (but not the on-die ECC metadata) values. Second, a reactive profiling phase that safely identifies bits at risk of indirect errors using a secondary N-error-correcting ECC within the memory controller. The secondary ECC is used *only* for reactive profiling: upon identifying an error, the corresponding bit is marked as at-risk, which signals the repair mechanism to repair the bit thereafter.

Prior work [145] shows that knowing the on-die ECC's internal implementation (i.e., its parity-check matrix)[4] enables *calculating* which post-correction errors a given set of pre-correction errors can cause. Therefore, we introduce two HARP variants: HARP-Aware (HARP-A) and HARP-Unaware (HARP-U), which do and do not know the parity-check matrix, respectively. HARP-A does *not* improve coverage over HARP-U because it has no additional visibility into the pre-correction errors. However, HARP-A demonstrates that, although knowing the parity-check matrix alone does *not* overcome the three profiling challenges, it does provide a head-start for reactive profiling based on the results of active profiling.

We evaluate HARP in simulation relative to two state-of-the-art baseline error profiling algorithms: *Naive* (which represents the vast majority of previous-proposed profiling algorithms [11, 12, 24, 27, 29, 30, 40, 48, 76–79, 82, 84, 86, 87, 103, 104, 109–111, 132, 145, 147, 149, 150, 159, 168, 171, 183]) and *BEEP* [145]. We show that HARP quickly achieves coverage of *all bits at risk of direct errors* while Naive and BEEP are either slower or unable to achieve full coverage. For example, when there are 2/3/4/5 bits at risk of pre-correction error that each fail with probability 0.5, HARP[5] achieves 99th-percentile[6] coverage in 20.6%/36.4%/52.9%/62.1% of the profiling rounds required by the best baseline algorithm. Based on our evaluations, we conclude that HARP effectively overcomes the three profiling challenges. We publicly release our simulation tools as open-source software on Zenodo [148] and Github [3].

To demonstrate the end-to-end importance of having an effective profiling mechanism, we also perform a case study of how HARP, Naive, and BEEP profiling can impact the overall bit error rate of a system equipped with an ideal bit-repair mechanism that perfectly repairs all identified at-risk bits. We show that, because HARP achieves full coverage of bits at risk of direct errors, it enables the bit-repair mechanism to repair all errors. Although Naive eventually achieves full coverage, it takes substantially longer to do so (by 3.7× for a raw per-bit error probability of 0.75). In contrast, BEEP does *not* achieve full coverage, so the bit-repair mechanism is unable to repair all errors that occur during system operation.

We make the following contributions:

- We conduct the first analytical study of how on-die ECC affects system-level bit-granularity error profiling. We identify three key challenges that must be addressed to enable practical and effective error profiling in main memory chips with on-die ECC.
- We introduce Hybrid Active-Reactive Profiling (HARP), a new bit-granularity error profiling algorithm that quickly achieves full coverage of at-risk bits by profiling errors in two phases: (1) active profiling using raw data bit values to efficiently identify all bits at risk of direct errors; and (2) reactive profiling with the guarantee that all remaining at-risk bits can be safely identified.
- We introduce two HARP variants, HARP-Unaware (HARP-U) and HARP-Aware (HARP-A). HARP-A exploits knowledge of the on-die ECC parity-check matrix to achieve full coverage of at-risk bits faster by precomputing at-risk bit locations.
- We evaluate HARP relative to two baseline profiling algorithms. An example result shows that HARP achieves 99th-percentile coverage of all bits at risk of direct error in 20.6%/36.4%/52.9%/62.1% of the profiling rounds required by the best baseline technique given 2/3/4/5 pre-correction errors.
- We present a case study of how HARP and the two baseline profilers impact the overall bit error rate of a system equipped with an ideal repair mechanism that perfectly repairs all identified at-risk bits. We show that HARP enables the repair mechanism to mitigate all errors faster than the best-performing baseline (e.g., by 3.7× for a raw per-bit error probability of 0.75).

2 BACKGROUND AND MOTIVATION

This section briefly discusses repair mechanisms, error profiling, our assumed error model, and on-die ECC. For more detailed background information, we refer the reader to other publications on repair [55, 61, 92, 93, 109, 112, 136, 157, 158, 168, 176, 179], profiling [11, 27, 30, 40, 48, 76–79, 87, 103, 111, 147, 150, 159, 171, 183] and main memory error mechanisms [9, 25, 28, 46, 47, 62, 73, 77, 87, 88, 91, 93, 101, 105, 110, 120, 134, 136, 152, 170, 178].

2.1 Addressing Scaling-Related Errors

Continual increases to memory storage density exacerbate various technology-specific error mechanisms (e.g., DRAM data retention [21, 46, 47, 54, 74, 76, 77, 93, 110, 136, 144, 147, 161, 162, 171]) that result in increasing error rates. To address these errors, main memory manufacturers have already begun to use on-die ECC (e.g.,

[3]Nonsystematic designs require additional decoding effort (i.e., more logic operations) because data *cannot* be read *directly* from stored values [181]. This increases the energy consumption of read operations and either reduces the overall read timing margins available for other memory operations or increases the memory read latency.

[4]Potentially provided through manufacturer support, datasheet information, or previously-proposed reverse engineering techniques [145].

[5]HARP-U and HARP-A have identical coverage of bits at risk of direct error.

[6]We report 99th percentile coverage to compare against baseline configurations that do not achieve full coverage within the maximum simulated number of profiling rounds (due to simulation time constraints, as discussed in §7.1.2).

LPDDR4 [98, 99, 121, 140], DDR5 [67], STT-RAM [39]) as a black-box error mitigation technique within the memory chip, regardless of whether or not it is the most efficient solution for a given system. On-die ECC addresses uncorrelated single-bit errors that limit a manufacturers' factory yield [21, 43, 60, 74, 121, 145, 146, 162] and is already prevalent among commodity DRAM chips today. Therefore, it is imperative that system-level error-mitigation mechanisms take on-die ECC into account, as clearly motivated by several prior works [21, 32, 43, 69, 137, 145, 162].

2.1.1 Mitigating High Error Rates. As memory technologies continue to scale, prior works [21, 92, 109, 135, 136, 162] argue that raw bit error rates will grow to very high values (e.g., $> 10^{-4}$ in DRAM [21, 136]).[7] Enabling robust memory operation at these error rates is an established research area for future DRAM and nonvolatile memories [34, 92, 93, 109, 112, 133, 135, 136, 157, 162, 174, 180] because it allows for both continued memory density scaling and enables new system operating points and use-cases (e.g., reliably reducing memory timings and voltages [22–24, 72, 84, 95, 103, 138, 160, 182]) that are not feasible today. In general, fine-grained (e.g., bit or word granularity) hardware repair mechanisms can mitigate high error rates more efficiently than conventional hardware error-mitigation techniques (e.g., ECC) [92, 93, 109, 112, 136, 150, 162]. §2.2 discusses repair mechanisms in further detail.

2.1.2 Consolidating In-Memory and Memory Controller Error Mitigations. Unlike Flash memory [18–20, 113, 114], main memory is generally designed separately from the memory controller [130]. Unfortunately, this separation discourages building a unified error-mitigation mechanism across the memory and its controller. This is exemplified by the widespread use of proprietary DRAM on-die ECC, which introduces new reliability challenges for designing error mitigation mechanisms within the DRAM controller [21, 32, 43, 137, 145, 162]. In general, the standardized interface between the memory and the controller (e.g., JEDEC DRAM standards [64, 67, 68]) must be modified to develop a joint solution, which impacts all manufacturers and consumers involved, and thus is a laborious and long (and often politically-charged) process. Therefore, we and other state-of-the-art hardware-based repair mechanisms focus on minimizing the changes required to the interface or memory chips themselves [80, 81, 109, 136, 150, 162].

2.1.3 Synergy with Other Error Mitigation Approaches. Effective main memory error management is a large research space with solutions spanning the entire hardware-software stack. There are many promising solution directions, including software-driven repair techniques such as page retirement [12, 58, 118, 120, 139, 171] and software-assisted techniques such as post-package repair (PPR) [21, 55, 64, 67, 74, 116, 136, 162]. Unfortunately, these mechanisms have limitations that make them ill-suited to address the high error rates that we target. For example, page retirement operates at a coarse (i.e., system memory page) granularity, so it both wastes significant capacity to repair the many errors at high error rates and cannot easily repair in-use pages [106, 118, 120]. PPR provides only a few spare rows (e.g., one per bank in DDR4 [81, 122]) and suffers from similar drawbacks as page retirement due to operating at a coarse granularity (i.e., DRAM row). In contrast, hardware-based repair mechanisms represent the state-of-the-art in addressing scaling-related main memory errors.

We note that other error mitigation techniques may be used synergistically with hardware-based repair, potentially to address different error models simultaneously. Our work both (1) identifies the challenges that on-die ECC introduces for repair mechanisms; and (2) demonstrates a concrete way (i.e., HARP) to address these challenges with minimal changes to existing systems. Given that on-die ECC is highly prevalent today (e.g., LPDDR4 [121, 140], DDR5 [67], STT-RAM [39]), our ideas are applicable to and evaluated based on current state-of-the-art solutions. In doing so, we believe our work will help guide future solutions that develop new abstractions to step beyond simple on-die ECC as it exists today.

2.2 Enabling Repair Alongside On-Die ECC

Repair mechanisms [55, 61, 84, 92, 93, 103, 109, 112, 116, 136, 157, 158, 162, 168, 176, 179] perform repair at granularities ranging from kilobytes to single bits. The granularity at which a repair mechanism identifies at-risk locations is its *profiling granularity*. For example, on-die row and column sparing [21, 55, 64, 67, 74, 116, 136, 162] requires identifying at-risk locations at (or finer than) the granularity of a single memory row. Table 1 categorizes key repair mechanisms based their profiling granularities. In general, coarse-grained repair requires less intrusive changes to the system datapath because repair operations can align with data blocks in the datapath (e.g., DRAM rows, cache lines, processor words). However, this means that the repair mechanism suffers from more internal fragmentation because each repair operation sacrifices more memory capacity regardless of how few bits are actually at risk of error.

Profiling Granularity	Size (Bits)	Examples
System page	32 K	RAPID [171], RIO [12], Page retirement [12, 58, 118, 139, 171]
DRAM external row	2-64 K	PPR [21, 55, 64, 67, 74, 116, 136, 162], Agnos [150], RAIDR [111], DIVA [103]
DRAM internal row/col	512-1024	Row/col sparing [21, 55, 74, 116, 136, 162], Solar [84]
Cache block	256-512	FREE-p [179], CiDRA [162]
Processor word	32-64	ArchShield [136]
Byte	8	DRM [61]
Single bit	1	ECP[8] [157], SECRET [109], REMAP [168], SFaultMap [92], HOTH [112], FLOWER [93], SAFER [158], Bit-fix [176]

Table 1: Survey of prevalent memory repair mechanisms.

Because of this tradeoff between intrusiveness and fragmentation, finer repair granularities are more efficient at higher error rates [21, 136, 162]. Fig. 2 illustrates this by showing the expected proportion of unnecessarily repaired bits (i.e., the amount of non-erroneous memory capacity that is sacrificed alongside truly erroneous bits due to internal fragmentation) (y-axis) at various raw bit error rates (x-axis) when mitigating uniform-random single-bit errors at different repair granularities. We see that coarse-grained repair becomes extremely wasteful as errors become more frequent, e.g., wasting over 99% of total memory capacity in the worst case for a 1024-bit granularity at a raw bit error rate of 6.8×10^{-3}. Note that the expected wasted storage decreases once the error rate is

[7] Exact error rate values of real memory chips are proprietary secrets because they can reveal manufacturing details and/or information relating to market competitiveness [42, 136]. Prior works draw reasonable estimates from circuit metrics, e.g., coefficient of variation [61, 93, 135].

[8] ECP corrects individual bits, but its pointer size can be adjusted to different granularities as required.

sufficiently high because an increasing proportion of bits become truly erroneous, which reduces the wasted bits for each repair operation. In contrast, bit-granularity repair (denoted with the line for '1') does *not* suffer from internal fragmentation. For this reason, repair mechanisms designed for higher error rates generally employ finer-granularity profiling and repair [92, 93, 109, 112].

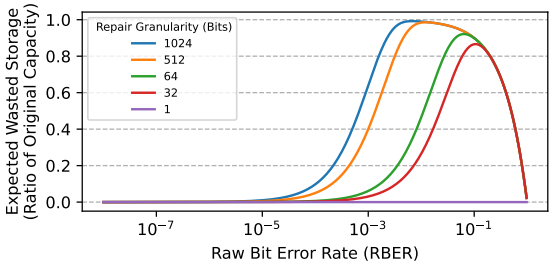


Figure 2: Expected amount of wasted storage capacity when repairing single-bit errors at various repair granularities.

2.3 Practical and Effective Error Profiling

Any repair mechanism's effectiveness strongly depends on the effectiveness of the error profiling algorithm that it uses because the repair mechanism can only repair memory locations that it knows are at risk of error. In this section, we define the key properties of practical and effective active and reactive profilers.

2.3.1 Active Profiling. Active profiling algorithms take exclusive control of a memory chip in order to (possibly destructively) test worst-case data and access patterns [11, 30, 40, 48, 77–79, 87, 103, 111, 147, 171, 183], so the system cannot perform useful work while profiling. Therefore, an active profiler must identify at-risk bits as quickly and comprehensively as possible. We quantify this by measuring the fraction of all at-risk bits that a given profiler identifies (i.e., its *coverage*) across a fixed number of testing rounds.

2.3.2 Reactive Profiling. Reactive profiling algorithms (e.g., ECC scrubbing [6, 27, 50, 134, 150, 159]) passively monitor error-detection mechanisms during normal system operation, so their performance and energy impact is relatively low and can be amortized across runtime [50, 150]. However, because reactive profilers operate during runtime, they must ensure that they can not only detect, but also correct any errors that occur. Any errors that are not detected and corrected by the reactive profiler risk introducing failures to the rest of the system. In our work, we quantify the error-mitigation capability of a reactive profiler in terms of ECC correction capability, which is well-defined for ECCs used in memory hardware design (i.e., linear block codes [55, 154, 155]).

2.4 Errors and Error Models

Our work assumes uncorrelated single-bit errors because recent experimental studies and repair efforts from academia [15, 112, 135, 136, 162], industry [74], and memory manufacturers themselves [21, 60, 61, 121, 144] focus on single-bit errors as the primary reliability challenge with increasing storage density. In particular, DRAM and STT-RAM manufacturers use on-die ECC specifically to combat these errors in recent high-density chips [21, 39, 43, 60, 74, 121, 140]. Therefore, we assume that errors exhibit the following properties:

(1) *Bernoulli process:* independent of previous errors.
(2) *Isolated:* independent of errors in other bits.
(3) *Data-dependent:* dependent on the stored data pattern.

To first order, this error model suits a broad range of error mechanisms that relate to technology scaling and motivate the use of bit-granularity repair, including DRAM data-retention [12, 46, 47, 70, 71, 77, 90, 92–94, 108, 110, 136, 147, 171, 175] and read disturbance [87, 91, 143]; PCM endurance, resistance drift, and write disturbance [62, 73, 88, 101, 105, 157]; and STT-RAM data retention, endurance, and read disturbance [9, 25, 28, 134, 152, 170]. We use DRAM data-retention errors in our evaluations as a well-studied and relevant example. However, a profiler is fundamentally agnostic to the underlying error mechanism; it identifies at-risk bits based on whether or not they are observed to fail during profiling. Therefore, our analysis applies directly to any error mechanism that can be described using the aforementioned three properties.

Correlated Errors. Prior DRAM studies show evidence of correlated errors [5, 120, 163–165, 167]. However, we are not aware of evidence that such errors are a first-order concern of modern DRAM technology scaling. Correlated errors often result from faults outside the memory array [120] and are mitigated using fault-specific error-mitigation mechanisms (e.g., write CRC [64, 100], chipkill [8, 37, 137] or even stronger rank-level ECC [64, 83, 167]).

Low-Probability Errors. Other main memory error mechanisms exist that do not conform to our model, including time-dependent errors such as DRAM variable retention time [76, 110, 124, 150, 153, 161, 178] and single-event upsets such as particle strikes [117]. In general, these errors are either (1) inappropriate to address using a profile-guided repair mechanism, e.g., single-event upsets that do not repeat; or (2) rare or unpredictable enough that no realistic amount of active profiling is likely to identify them, so they are left to reactive profiling for detection and/or mitigation [150]. Identifying low-probability errors is a general challenge for *any* error profiler and is an orthogonal problem to our work. Prior approaches to identifying low-probability errors during active (e.g., increasing the probability of error [147]) or reactive (e.g., periodic ECC scrubbing [10, 150]) profiling are complementary to our proposed techniques and can be combined with HARP (e.g., during the active profiling phase described in §6.2, or by strengthening the secondary ECC as described in §6.3.2) to help identify low-probability errors.

2.5 Block Codes and Syndrome Decoding

Typical on-die ECC implementations use linear block codes [21, 59, 69, 98, 99, 121, 140, 162] whose operation can be summarized using matrix arithmetic. In our work, we assume single-error correcting (SEC) Hamming codes [49] that are adopted in modern LPDDR4 [121, 140] and DDR5 [67] DRAM chips.[9] This section briefly summarizes the operation of an SEC Hamming code in the context of on-die ECC. For further information, we refer the reader to extensive literature on error-correction coding [31, 123, 154, 155].

2.5.1 Encoding and Decoding. A Hamming code with k data bits and p parity-check bits is defined by a $(k, k+p)$ generator matrix G and a $(p, k+p)$ parity-check matrix H such that $G \cdot H^T = \mathbf{0}$ within

[9] Our analysis can theoretically generalize to stronger block codes (e.g., double-error correcting BCH [16, 53]), but we leave such generalization to future work given that such codes are currently unlikely to be used in latency-sensitive memory chips.

the finite field $GF(2)$. Equation 1 gives example H and G matrices that define a $k = 4$ SEC Hamming code.

$$\mathbf{G^T} = \left[\begin{array}{c|c} 1\,0\,0\,0 & 1\,1\,1 \\ 0\,1\,0\,0 & 1\,1\,0 \\ 0\,0\,1\,0 & 1\,0\,1 \\ 0\,0\,0\,1 & 0\,1\,1 \end{array}\right] \qquad \mathbf{H} = \left[\begin{array}{c|c} 1\,1\,1\,0 & 1\,0\,0 \\ 1\,1\,0\,1 & 0\,1\,0 \\ 1\,0\,1\,1 & 0\,0\,1 \end{array}\right] \tag{1}$$

Fig. 3 illustrates how a system interfaces with a memory module whose chip(s) use on-die ECC. To encode a k-bit *dataword* d, the *ECC encoder* computes a $(k + p)$-bit *codeword* c as $c = G \cdot d$. Upon incurring raw bit error(s), we denote the erroneous codeword as c'. To decode c' into a post-correction dataword d', the *ECC decoder* performs *syndrome decoding*, where a *syndrome* s is calculated as $s = H \cdot c'$. If s is nonzero, one or more errors must be present, and the bit position of the detected error can then be identified as the particular column of H that matches s.

If an uncorrectable error occurs, s may inadvertently match a parity-check matrix column that does *not* correspond to an actual error. In this case, the ECC decoder might introduce an *additional* error in the decoded data, which we refer to as a *miscorrection*. Note that the memory controller may interface with one or more memory chips at a time, potentially spreading a single data block across multiple on-die ECC words. We discuss this further in §6.3.

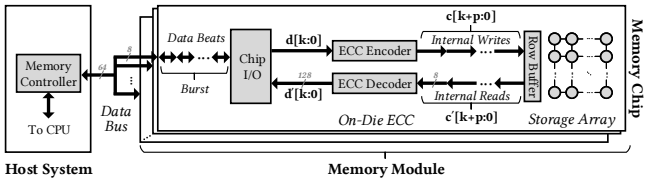


Figure 3: On-die ECC operation within a memory chip. Grey annotations show example bit-widths of data transfers.

2.5.2 Design Degrees of Freedom. When choosing H for a particular implementation, the designer is free to arrange its columns at will. Recent work [142] shows that some column arrangements can lead to more miscorrections than others. However, designers may choose column arrangements based on circuit latency, energy, or area concerns, regardless of each choice's effect on reliability. In our work, we assume that the code uses *systematic encoding*, which requires that H and G do not modify data bits during encoding (note the identity submatrices in Equation 1) but does not otherwise constrain the column arrangement. This encoding greatly simplifies the hardware encoding and decoding circuitry and is a realistic assumption for low-latency main memory chips [181].

3 FORMALIZING ERROR PROFILING

We express error profiling as a statistical process to understand the effects of on-die ECC. To this end, we first formalize the concepts of errors and error profiling. Then, we examine how on-die ECC changes the way that errors appear outside of the memory chip.

3.1 Representing the Probability of Error

We model memory as a one-dimensional bit-addressable array with address space $\mathcal{A}$. To describe errors within this address space, we define two Boolean random variables D_i and E_i that represent the data stored in bit $i \in \mathcal{A}$ and whether or not the bit experiences an error, respectively. Based on our discussion in §2.4, we model E_i as a Bernoulli random variable that is independent of $E_{j \neq i}$ but dependent on the data D_i. In general, E_i can depend on the data stored in other cells $D_{j \neq i}$, which expresses how a bit's probability of error changes with the data stored in nearby cells. Equation 2 shows the resulting probability mass function, parameterized by p, the probability that the bit will experience an error.

$$P(E_i = x | D_i, D_j, \cdots) = \begin{cases} p(D_i, D_j, \cdots) & \text{if } x = 1 \\ 1 - p(D_i, D_j, \cdots) & \text{if } x = 0 \end{cases} \tag{2}$$

In general, each bit has its own value of p depending on its intrinsic error characteristics. For example, prior work [147] experimentally demonstrates that p values are normally distributed across different bits for DRAM data-retention errors, i.e., $p \sim N(\mu, \sigma^2)$, with some normal distribution parameters $\{\mu, \sigma\}$ that depend on the particular memory chip and operating conditions such as temperature.

3.2 Incorporating On-Die ECC

With on-die ECC, we adjust our address space representation to include both *logical* bit addresses $\mathcal{A}$ as observed by the memory controller and *physical* bit addresses $\mathcal{B}$ within the memory storage array. In general, $|\mathcal{B}| > |\mathcal{A}|$ because $\mathcal{B}$ includes addresses for parity-check bits that are *not* visible outside of the memory chip. Next, we introduce two additional Boolean random variables: D_a (for dataword) and C_b (for codeword) that refer to the data values of logical bit $a \in \mathcal{A}$ and physical bit $b \in \mathcal{B}$ (i.e., before and after ECC encoding), respectively. Boolean variables E_a and R_b represent whether logical bit a and physical bit b experience post- and pre-correction errors, respectively. Note that E and D represent the same information from §3.1. We use C' and D' to refer to codeword and dataword values, respectively, that may contain errors.

On-die ECC determines D' from C' through syndrome decoding (described in §2.5) using the ECC parity-check matrix H comprised of columns $H[k + p : 0]$. The error syndrome is computed as $s = H[k + p : 0] \cdot R[k + p : 0]$ (referred to as $H \cdot R$ to simplify notation). Then, if s matches the i'th column $H[i]$, the ECC decoder flips the bit at position i. Given that H is systematically encoded (discussed in §2.5.1), $c[k : 0]$ is equal to $d[k : 0]$. Therefore, a post-correction error E_i (i.e., a mismatch between d_i and d'_i) can only occur in two cases: (1) an uncorrected raw bit error at position i (i.e., $R_i \wedge s \neq H[i]$); or (2) a miscorrection at position i (i.e., $\neg R_i \wedge s = H[i]$.) We refer to these two cases as *direct* and *indirect* errors, respectively. Equation 3 summarizes both cases that lead to a post-correction error.

$$P(E_i) = P(R_i \veebar H \cdot R = H[i]) \tag{3}$$

Equation 3 shows that bit i's probability of error depends *not only* on that of its encoded counterpart R_i, but *also* on those of *all other codeword bits* $R[k + p : 0]$. This means that on-die ECC introduces statistical dependence between *all bits* in a given ECC word through their mutual dependence on R. Furthermore, just as described in §3.1, R_i itself depends on the data value stored in cell i (i.e., C_i). As a result, bit i's probability of error depends on *both* the data values and the pre-correction errors present throughout the codeword.

Consequently, a given post-correction error E_i may *only* occur when a particular combination of pre-correction errors occurs. This is different from the case without on-die ECC, where E_i does *not* depend on the data or error state of any other bit $j \neq i$. We conclude

that on-die ECC transforms statistically independent pre-correction errors into ECC-dependent, correlated post-correction errors.

4 ON-DIE ECC'S IMPACT ON PROFILING

On-die ECC breaks the simple and intuitive assumption that profiling for errors is the same as profiling each bit individually. In this section, we identify three key challenges that on-die ECC introduces for bit-granularity profiling.

4.1 Challenge 1: Combinatorial Explosion

§3.2 shows that the position of an indirect error depends on the locations of all pre-correction errors R. This means that different uncorrectable patterns of pre-correction errors can cause indirect errors in different bits. In the worst case, *every unique combination* of pre-correction errors within a set of at-risk bits can lead to different indirect errors. This means that the set of bits that are at risk of post-correction errors is *combinatorially larger* than the set of bits at risk of pre-correction error.

As a concrete example, Fig. 4 shows a violin plot of each at-risk bit's per-bit probability of error (y-axis) when the codeword contains a fixed number of bits at risk of pre-correction errors (x-axis) that each fail with probability 0.5. Each violin shows the distribution (median marked in black) of per-bit error probabilities when simulating 70,000 ECC words for each of 1600 randomly-generated (71, 64) Hamming code parity-check matrices assuming a data pattern of 0xFF. We make two observations. First, the pre-correction error probabilities are all 0.5 (by design). This means that, without on-die ECC, all bits at risk of pre-correction error are easy to identify, i.e., each bit will be identified with probability $p = 1 - 0.5^N$ given N profiling rounds. For large N (e.g., $N > 10$, where $p > 0.999$), the vast majority of bits will be identified.

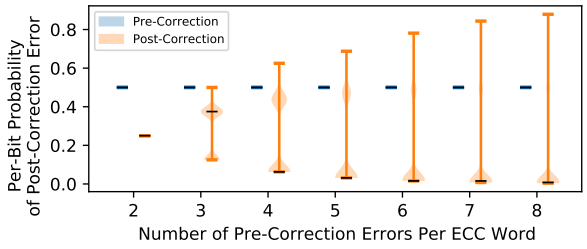


Figure 4: Distribution of each at-risk bit's error probability before and after application of on-die ECC.

Second, in contrast, the per-bit probabilities of post-correction error exhibit a wide range. However, the probability density for each violin is tightly concentrated at $Y \approx 0.4$ for $X = 3$ and shifts towards $Y = 0$ as the number of pre-correction errors increases. This means that the bits at risk of post-correction errors become *harder to identify* because they fail less often.

Table 2 shows the maximum number of bits at risk of post-correction errors that can be caused by a fixed number of bits at risk of pre-correction errors. This illustrates the worst-case scenario, where *every uncorrectable combination* of pre-correction errors (i.e., *pre-correction error pattern*) causes a unique indirect error. We see that n bits at risk of pre-correction errors can cause $2^n - 1$ unique pre-correction error patterns. Of these, n are correctable error patterns, leaving $2^n - n - 1$ *uncorrectable pre-correction error patterns*. Assuming that each of these patterns introduces a unique indirect error, the combination of bits at risk of direct and indirect error leads to $2^n - 1$ bits at risk of post-correction errors. Therefore, we conclude that on-die ECC exponentially increases the number of at-risk bits that the profiler must identify.

Bits at risk of pre-correction errors	n	**1**	**2**	**3**	**4**	**8**
Unique pre-correction error patterns	$2^n - 1$	1	3	7	15	255
Uncorrectable pre-correction error patterns	$2^n - n - 1$	0	2	4	11	247
Bits at risk of post-correction errors	$2^n - 1$	**1**	**3**	**7**	**15**	**255**

Table 2: On-die ECC amplifies a few bits at risk of pre-correction errors into exponentially many bits at risk of post-correction errors.

4.2 Challenge 2: Profiling without Feedback

Without on-die ECC, an at-risk bit is identified when the bit fails. This means that every profiling round provides useful feedback about which bits *are* and *are not* at risk of error. Unfortunately, with on-die ECC, a bit at risk of post-correction errors can *only* be identified when *particular combination(s)* of pre-correction errors occur. This has two negative consequences.

First, because the profiler cannot observe pre-correction errors, it does not know whether a particular combination of pre-correction errors has been tested yet. Therefore, the profiler *cannot* draw meaningful conclusions from observing a bit *not* to fail. Instead, the profiler must pessimistically suspect every bit to be at risk of post-correction errors, even after many profiling rounds have elapsed without observing a given bit fail. Second, each ECC word can only exhibit *one* pre-correction error pattern at a time (i.e., during any given profiling round). This serializes the process of identifying any two bits at risk of post-correction errors that fail under different pre-correction error patterns.

As a result, no profiler that identifies at-risk bits based on observing post-correction errors can quickly identify all bits at risk of post-correction errors. We refer to this problem as *bootstrapping* because the profiler must explore different pre-correction error patterns without knowing which patterns it is exploring. In §7.2.2, we find that bootstrapping limits the profiler's coverage of at-risk bits to incremental improvements across profiling rounds.

4.3 Challenge 3: Multi-Bit Data Patterns

Designing data patterns that induce worst-case circuit conditions is a difficult problem that depends heavily on the particular circuit design of a given memory chip and the error mechanisms it is susceptible to [26, 33, 36, 75, 125, 126]. Without on-die ECC, a bit can fail only in one way, i.e., when it exhibits an error. Therefore, the worst-case pattern needs to only consider factors that affect the bit itself (e.g., data values stored in the bit and its neighbors).

Unfortunately, with on-die ECC, a given post-correction error can potentially occur with multiple different pre-correction error patterns. Therefore, the worst-case data pattern must both (1) account for different ways in which the post-correction error can occur; and (2) for each way, consider the worst-case conditions for the individual pre-correction errors to occur simultaneously. This is a far more complex problem than without on-die ECC [41, 44], and in general, there may not even be a single worst-case data pattern

that exercises all possible cases in which a given post-correction error might occur. To our knowledge, no prior work has developed a general solution to this problem, and we identify this as a key direction for future research.

5 ADDRESSING THE THREE CHALLENGES

We observe that all three profiling challenges stem from the *lack of access* that the profiler has into pre-correction errors. Therefore, we conclude that *some* transparency into the on-die ECC mechanism is necessary to enable practical error profiling in the presence of on-die ECC. This section discusses options for enabling access to pre-correction errors and describes our design choices for HARP.

5.1 Necessary Amount of Transparency

To reduce the number of changes we require from the memory chip, we consider the minimum amount of information that the profiler needs to make error profiling *as easy as* if there were no interference from on-die ECC. To achieve this goal, we examine the following two insights that are derived in §3.2.

(1) Post-correction errors arise from either direct or indirect errors.
(2) The number of concurrent indirect errors is limited to the correction capability of on-die ECC.

First, we observe that it is *not* necessary for the profiler to have full transparency into the on-die ECC mechanism or pre-correction errors. If all bits at risk of direct errors can be identified, all remaining indirect errors are upper-bounded by on-die ECC's correction capability. Therefore, the indirect errors can be safely identified from within the memory controller, e.g., using a reactive profiler.

Second, we observe that the profiler can determine exactly which pre-correction errors occurred within the data bits (though not the parity-check bits) simply by knowing at which bit position(s) on-die ECC performed a correction operation. This is because the data bits are systematically encoded (as explained in §2.5.2), so their programmed values must match their encoded values. By observing which bits experienced direct error(s), the profiler knows which pre-correction errors occurred within the encoded data bits.

Based on these observations, we require that the profiler be able to identify which direct errors occur on every access, including those that on-die ECC corrects. Equivalently, on-die ECC may expose the error-correction operation that it performs so that the profiler can infer the direct errors from the post-correction data.

5.2 Exposing Direct Errors to the Profiler

We consider two different ways to inform the profiler about pre-correction errors within the data bits.

(1) *Syndrome on Correction.* On-die ECC reports the error syndrome calculated on all error correction events, which corresponds to the bit position(s) that on-die ECC corrects.
(2) *Decode Bypass.* On-die ECC provides a read access path that bypasses error correction and returns the raw values stored in the data portion of the codeword.

We choose to build upon decode bypass because we believe it to be the easiest to adopt for three key reasons. First, there exists precedent for similar on-die ECC decode bypass paths from both academia [43] and industry [13] with trivial modifications to internal DRAM hardware, and an on-die ECC disable configuration register is readily exposed in certain DRAM datasheets [7]. Second, prior works already reverse-engineer both the on-die ECC algorithm [145] and raw bit error rate [146] without access to raw data bits or insight into the on-die ECC mechanism, so we do not believe exposing a decode bypass path reveals significantly more sensitive information. Third, we strongly suspect that such a bypass path *already exists* for post-manufacturing testing [173]. This is reasonable because systematically-encoded data bits can be read out directly without requiring further transformation. If so, exposing this capability as a feature would likely require minimal engineering effort for the potential gains of new functionality. However, we recognize that the details of the on-die ECC implementation depend on the particular memory chip design, and it is ultimately up to the system designer to choose the most suitable option for their system.

5.3 Applicability to Other Systems

Any bit-granularity profiler operating without visibility into the pre-correction errors suffers form the three profiling challenges we identify in this work. Even a hypothetical future main memory system whose memory chips and controllers are designed by the same (or two trusted) parties will need to account for and overcome these profiling challenges when incorporating a repair mechanism that relies on practical and effective profiling.

6 HYBRID ACTIVE-REACTIVE PROFILING

We introduce the Hybrid Active-Reactive Profiling (HARP) algorithm, which overcomes the three profiling challenges introduced by on-die ECC discussed in §5. HARP separates profiling into *active* and *reactive* phases that independently identify bits at risk of *direct* and *indirect* errors, respectively.

6.1 HARP Design Overview

Fig. 5 illustrates the high-level architecture of a HARP-enabled system, with the required error-mitigation resources shown in blue. The memory chip exposes a read operation with the ability to bypass on-die ECC and return the raw data (though not parity-check) bits. The memory controller contains a repair mechanism with an associated error profile alongside both an *active* and *reactive* profiler. During active profiling, the active profiler uses the ECC bypass path to search for bits at risk of direct errors. Because the active profiler interfaces *directly* with the raw data bits, its profiling process is equivalent to profiling a memory chip without on-die ECC. If and when direct errors are observed, the active profiler communicates their locations to the repair mechanism's error profile.

After active profiling is complete, the reactive profiler (i.e., a secondary ECC code with correction capability at least as strong as that of on-die ECC) continuously monitors for bits at risk of indirect errors. The reactive profiler is responsible *only* for identifying bits at risk of indirect errors the first time that they fail. If and when the reactive profiler identifies an indirect error, the location of the error is recorded to the error profile for subsequent repair.

6.2 Active Profiling Implementation

The active profiler follows the general round-based algorithm employed by state-of-the-art error profilers, as discussed in §1. Each

round of testing first programs memory cells with a standard memory data pattern that is designed to maximize the chance of observing errors (e.g., $0xFF$, $0x00$, random data) [4, 77, 87, 110, 126]. Patterns may or may not change across testing rounds depending on the requirements of the particular data pattern. Once sufficiently many rounds are complete, the set of at-risk bits identified comprises the union of all bits identified across all testing rounds.

We assume that the active profiler achieves full coverage of bits at risk of direct errors by leveraging any or all of the worst-case testing techniques developed throughout prior works [77, 79, 126, 127, 147]. This is feasible because the active profiler can read and write to the raw data bits exploiting the ECC bypass path and the systematically-encoded data bits, respectively. Therefore, the active profiler can use techniques developed for memory chips without on-die ECC.

6.3 Reactive Profiling Implementation

HARP requires that the secondary ECC have correction capability *at least* as high as the number of indirect errors that on-die ECC can cause at one time. This requires the layout of secondary ECC words to account for the layout of on-die ECC words: the two must combine in such a way that every on-die ECC word is protected with the necessary correction capability by the secondary ECC. For example, with a single-error correcting on-die ECC that uses 128-bit words, the memory controller must ensure that every 128-bit on-die ECC word is protected with *at least* single-error correction.

How this is achieved depends heavily on a given system's memory architecture. For example, depending on the size of an on-die ECC word and how many memory chips the memory controller interfaces with, on-die ECC words may be split across different data transfers. In this case, the system designer must choose a design that matches their design goals, e.g., dividing secondary ECC words across multiple transfers (which introduces its own reliability challenges [43]), or interleaving secondary ECC words across multiple on-die ECC words (which could require stronger secondary ECC).

Without loss of generality, we assume that the memory controller interfaces with a single memory chip at a time (e.g., similar to some LPDDR4 systems [65]) and provisions a single-error correcting code per on-die ECC word. Such a system is sufficient for demonstrating the error profiling challenges that we address in this work. Matching the granularity of secondary and on-die ECC words for arbitrary systems is *not* a problem unique to our work since *any* secondary ECC that is designed to account for the effects of on-die ECC must consider how the two interact [21, 43]. Therefore, we leave a general exploration of the tradeoffs involved to future work.

***6.3.1 HARP-U and HARP-A.*.** We introduce two variants of HARP: HARP-A and HARP-U, which are aware and unaware of the on-die ECC parity-check matrix H, respectively. HARP-A uses this knowledge to *precompute*[10] bits at risk of indirect error given the bits at risk of direct error that are identified during active profiling. HARP-A does not provide benefits over HARP-U during active profiling. However, HARP-A reduces the number of bits at risk of indirect error that remain to be identified by reactive profiling.

6.3.2 Increasing the Secondary ECC Strength. The secondary ECC is used for reactive profiling and must provide equal or greater correction capability than on-die ECC to safely identify indirect errors. Current on-die ECC designs are limited to simple single-error correcting codes due to area, energy, and latency constraints within the memory die [43, 121, 162], so the secondary ECC can be correspondingly simple. However, if either (1) future on-die ECC designs become significantly more complex; or (2) the system designer wishes to address other failure modes (e.g., component-level failures) using the secondary ECC, the system designer will need to increase the secondary ECC strength accordingly. Whether profile-based repair remains a feasible error-mitigation strategy in this case is ultimately up to the system designer and their reliability goals, so we leave further exploration to future work.

6.4 Limitations

HARP relies on the active profiler to achieve full coverage of bits at risk of direct errors so that the reactive profiler *never* observes a direct error (i.e., the reactive profiler's correction capability is never exceeded). Consequently, if the active profiler fails to achieve full coverage, the reactive profiler may experience indirect errors *in addition to* direct error(s) missed by active profiling.

We acknowledge this as a theoretical limitation of HARP, but we do not believe it restricts HARP's potential impact to future designs and scientific studies. This is because achieving full coverage of at-risk bits without on-die ECC is a long-standing problem that is complementary to our work. Prior works have studied this problem in detail [76, 79, 147, 150], and any solution developed for chips without on-die ECC can be immediately applied to HARP's active profiling phase, effectively reducing the difficulty of profiling chips with on-die ECC to that of chips without on-die ECC.

7 EVALUATIONS

In this section, we study how HARP's coverage of direct and indirect errors changes with different pre-correction error counts and per-bit error probabilities to both (1) demonstrate the effect of the three profiling challenges introduced by on-die ECC and (2) show that HARP overcomes the three challenges.

[10] Using the methods described in detail in prior work [145].

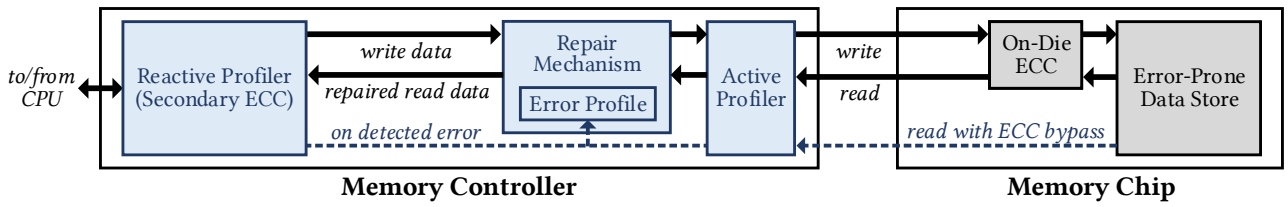


Figure 5: Block diagram summarizing the error-mitigation resources (in blue) of a HARP-enabled system.

7.1 Evaluation Methodology

We evaluate error coverage in simulation because, unlike with a real device, we can accurately compute error coverage using precise knowledge of which errors are and are not possible. This section describes our simulation methodology.

7.1.1 Baselines for Comparison. We compare HARP-U and HARP-A with two baseline profiling algorithms that use multiple rounds of testing with different data patterns to identify at-risk bits based only on observing post-correction errors.

(1) *Naive*, which represents the vast majority of prior profilers that operate without knowledge of on-die ECC [11, 12, 24, 27, 29, 30, 40, 48, 76–79, 82, 84, 86, 87, 103, 104, 109–111, 132, 145, 147, 149, 150, 159, 168, 171, 183] (described in §6.2).

(2) *BEEP*, the profiling algorithm supported by the reverse-engineering methodology BEER [145]. BEEP carefully constructs data patterns intended to systematically expose post-correction errors based on having reverse-engineered the on-die ECC parity-check matrix. We follow the SAT-solver-based methodology as described by [145] and use a random data pattern before the first post-correction error is confirmed.

7.1.2 Simulation Strategy. We extend the open-source DRAM on-die ECC analysis infrastructure released by prior work [1, 145] to perform Monte-Carlo simulations of DRAM data retention errors. We release our simulation tools on Zenodo [148] and Github [3]. We simulate error injection and correction using single-error correcting Hamming codes [49] representative of those used in DRAM on-die ECC (i.e., (71, 64) [59, 137] and (136, 128) [98, 99, 121, 140] code configurations). All presented data is shown for a (71, 64) code, and we verified that our observations hold for longer (136, 128) codes. We simulate 1,036,980 total ECC words across 2769 randomly-generated parity-check matrices over ≈20 days of simulation time (discussed in §A.8.2). For each profiler configuration, we simulate 128 profiling rounds because this is enough to understand the behavior of each configuration (e.g., the shapes of each curve in Fig. 6), striking a good balance with simulation time.

We inject errors according to the model discussed in §2.4 to simulate the effect of uniform-random, data-dependent errors. We assume that all bits are true-cells [96, 110] that experience errors *only* when programmed with data '1', which is consistent with experimental observations made by prior work [96, 145]. To study how varying error rates impact profiling, we simulate bit errors with Bernoulli probabilities of 1.0, 0.75, 0.5, and 0.25 and separate our results based on the total number of pre-correction errors n injected into a given ECC word. Using this approach, one can easily determine the effect of an arbitrary raw bit error rate by summing over the individual per-bit error probabilities.

We define *coverage* as the proportion of all at-risk bits that are identified. We calculate coverage using the Z3 SAT solver [35], computing the total number of post-correction errors that are possible for a given (1) parity-check matrix; (2) set of pre-correction errors; and (3) (possibly empty) set of already-discovered post-correction errors. Note that a straightforward computation of coverage given on-die ECC is extremely difficult for data-dependent errors: each data pattern programs the parity-check bits differently, thereby provoking different pre-correction error patterns. Therefore, using the SAT solver, we accurately measure the bit error rate of *all possible* at-risk bits across *all possible* data patterns.

We simulate three different data patterns to exercise data-dependent behavior: `random`, `charged` (i.e., all bits are '1's), and `checkered` (i.e., consecutive bits alternate between '0' and '1'). For the `random` and `checkered` data patterns, we invert the data pattern each round of profiling. For the `random` pattern, we change the random pattern after every two profiling rounds (i.e., after both the pattern and its inverse are tested). We ensure that each profiler is evaluated with the exact same set of ECC words, pre-correction error patterns, and data patterns in order to preserve fairness when comparing coverage values. Unless otherwise stated, all data presented uses the `random` pattern, which we find performs on par or better than the static `charged` and `checkered` patterns that do *not* explore different pre-correction error combinations.

7.2 Active Phase Evaluation

We study the number of profiling rounds required by each profiling algorithm to achieve coverage of direct errors. We omit HARP-A because its coverage of direct errors is equal to that of HARP-U.

7.2.1 Direct Error Coverage. Fig. 6 shows the coverage of bits at risk of direct errors that each profiler cumulatively achieves (y-axis) over 128 profiling rounds (x-axis) assuming four different values of pre-correction errors per ECC word (2, 3, 4, and 5). We report results for four different per-bit error probabilities of the injected pre-correction errors (25%, 50%, 75%, 100%). For each data point, we compute coverage as the proportion of at-risk bits identified out of all at-risk bits across all simulated ECC words.

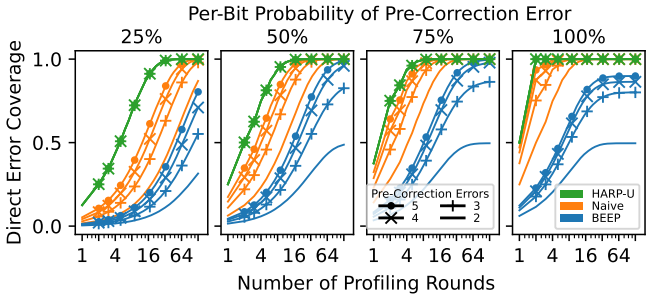


Figure 6: Coverage of bits at risk of direct errors.

We make two observations. First, HARP consistently and quickly achieves full coverage, *regardless of* the number or per-bit error probabilities of the injected pre-correction errors. This is because HARP bypasses on-die ECC correction, identifying each at-risk bit independently, regardless of which error occurs in which testing round. In contrast, both Naive and BEEP (1) require more testing rounds to achieve coverage parity with HARP and (2) exhibit significant dependence on the number of pre-correction errors. This is a direct result of on-die ECC: each post-correction error depends on particular combination(s) of pre-correction errors, and achieving high coverage requires these combinations to occur in distinct testing rounds. We conclude that that HARP effectively overcomes the first profiling challenge by directly observing pre-correction errors, while Naive and BEEP must both rely on uncorrectable error patterns to incrementally improve coverage in each round.

Second, although, HARP and Naive both *eventually* achieve full coverage, BEEP *fails* to do so in certain cases. This is because BEEP does not explore all pre-correction error combinations necessary to expose each bit at risk of direct errors. We attribute this behavior to a nuance of the BEEP algorithm: BEEP crafts data patterns that increase the likelihood of indirect errors. Unfortunately, these patterns are slow to explore different combinations of pre-correction errors, which leads to incomplete coverage. This is consistent with prior work [145], which finds that BEEP exhibits low coverage when pre-correction errors are sparse or occur with low probability. We find that Naive also fails to achieve full coverage when using static data patterns (e.g., `checkered`) for the same reason.

7.2.2 Bootstrapping Analysis. Fig. 7 shows the distribution (median marked with a horizontal line) of the number of profiling rounds required for each profiler to observe *at least one* direct error in each ECC word. If no post-correction errors are identified, we conservatively plot the data point as requiring 128 rounds, which is the maximum number of rounds evaluated (discussed in §7.1.2). The data illustrates the difficulty of bootstrapping because observing any post-correction error with on-die ECC requires a specific combination of pre-correction errors to occur.

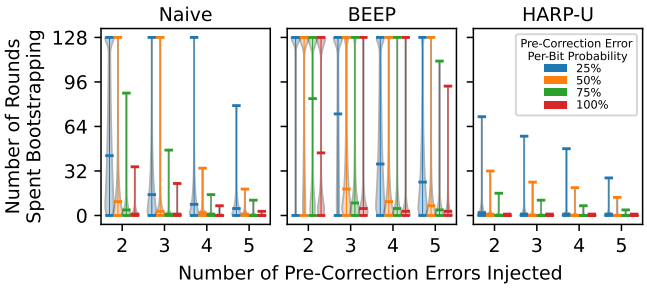


Figure 7: Distribution of the number of profiling rounds required to identify the first direct error across all simulated ECC words.

We make three observations. First, we see that HARP identifies the first error far more quickly than Naive or BEEP profiling across *all* configurations. Second, HARP never fails to identify at least one error within 128 rounds. Third, in contrast, BEEP sometimes cannot identify an error at all due to a combination of (1) the low per-bit pre-correction error probability and (2) the bootstrapping problem (i.e., more testing rounds does *not* guarantee higher coverage unless those rounds explore different uncorrectable patterns). We conclude that HARP effectively addresses the bootstrapping challenge by directly observing pre-correction errors instead of relying on exploring different uncorrectable error patterns.

7.3 Reactive Phase Evaluation

In this section, we study each error profiler's coverage of bits at risk of indirect errors and examine the correction capability required from the secondary ECC to safely identify the at-risk bits remaining after active profiling. It is important to note that, unlike HARP, neither Naive nor BEEP profiling achieve full coverage of bits at risk of *direct* errors for all configurations. In such cases, multi-bit errors can occur during reactive profiling that are not safely identified by a single-error correcting code (studied in §7.3.2), regardless of the profiler's coverage of bits at risk of indirect errors.

7.3.1 Indirect Error Coverage. Fig. 8 shows the proportion of all bits that are at risk of indirect errors that each profiler *has missed* per ECC word throughout 128 rounds of profiling. This is equivalent to the number of at-risk bits that reactive profiling has to identify. We evaluate an additional configuration, HARP-A+BEEP, which employs BEEP to identify the remaining at-risk bits once HARP-A has identified all bits at risk of direct errors.

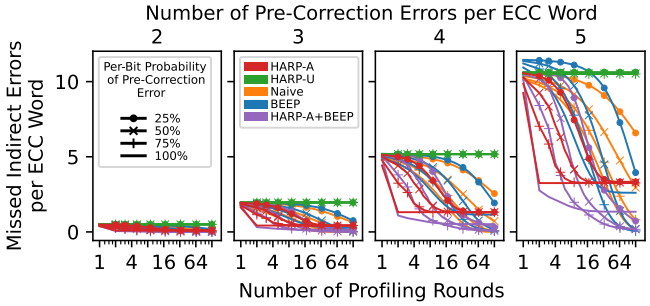


Figure 8: Coverage of bits at risk of indirect errors.

We make three observations. First, HARP-U does not identify any bits at risk of indirect errors[11] because it bypasses the on-die ECC correction process that causes indirect errors. In contrast, HARP-A quickly identifies a subset of all bits at risk of indirect errors by predicting them from the identified direct errors. Note that HARP-A cannot identify *all* bits at risk of indirect errors because doing so would require knowing which parity-check bits are at risk of error, which the on-die ECC bypass path does not reveal.

Second, combining HARP-A with BEEP effectively overcomes HARP-A's inability to identify pre-correction errors within the parity-check bits. This is because HARP-A+BEEP synergistically combines (1) HARP's ability to quickly identify bits at risk of direct errors with (2) BEEP's ability to exploit *known* at-risk bits to *expose* others. The combined configuration quickly identifies bits at risk of indirect errors, achieving coverage similar to Naive and BEEP profiling in less than half the number of profiling rounds.

Third, both Naive and BEEP achieve relatively high coverage of indirect errors after many (i.e., > 64) rounds compared to HARP-U and HARP-A. This is because both Naive and BEEP continually explore different uncorrectable error patterns, steadily exposing more and more indirect errors. BEEP achieves higher coverage because its algorithm deliberately seeks out pre-correction error combinations that are more likely to cause post-correction errors, thereby exposing more indirect errors in the long run.

We conclude that knowing the on-die ECC parity-check matrix helps HARP-A and BEEP identify bits at risk of indirect errors, thereby reducing the number of indirect errors that must be identified by by the secondary ECC during reactive profiling.

7.3.2 Secondary ECC Correction Capability. Fig. 9 shows the worst-case (i.e., maximum) number of post-correction errors that can occur simultaneously within an ECC word after active profiling.

[11] Except for a small number of direct and indirect errors that overlap.

This number is the correction capability required from secondary ECC to safely perform reactive profiling.

Maximum Error Count. Fig. 9a shows a normalized histogram of the maximum number of post-correction errors that can occur simultaneously within each simulated ECC word given all at-risk bits missed after 128 rounds of active profiling. We observe that both HARP-U and HARP-A exhibit *at most one* post-correction error across all configurations. This is because HARP identifies all bits at risk of direct errors within 128 profiling rounds (shown in Fig. 6), so only one error may occur at a time (i.e., an indirect error). In contrast, both Naive and BEEP are susceptible to multi-bit errors. In particular, BEEP's relatively low coverage of bits at risk of direct errors means that many multi-bit error patterns remain possible. We conclude that, after 128 rounds of active profiling, a single-error correcting secondary ECC is *sufficient* to perform reactive profiling for HARP but *insufficient* to do so for Naive and BEEP.

Maximum Error Count. Fig. 9b shows how many active profiling rounds are required to ensure that *no more than* an x-axis value of post-correction errors can occur simultaneously in a single ECC word during reactive profiling. We conservatively report results for the 99th percentile of all simulated ECC words because neither Naive nor BEEP achieve full coverage of bits at risk of direct errors for all configurations within 128 profiling rounds. In cases where 128 profiling rounds are insufficient to achieve 99th-percentile values, we align the bar with the top of the plot.

We make two observations. First, both HARP configurations perform *significantly* faster than Naive and BEEP. For example, with a 50% pre-correction per-bit error probability, HARP ensures that no more than one post correction error can occur in 20.6%/36.4%/52.9%/62.1% of the profiling rounds required by Naive given 2/3/4/5 pre-correction errors. This is because HARP quickly identifies all bits at risk of direct errors, while Naive and BEEP both either (1) take longer to do so; or (2) fail to do so altogether (e.g., for the 100th percentile at a 50% per-bit error probability). Second, BEEP performs much worse than any other profiler because it exhibits extremely low coverage of bits at risk of direct error (studied in §7.2.1). We conclude that achieving high coverage of bits at risk of direct errors is essential for minimizing the correction capability of the secondary ECC.

7.4 Case Study: DRAM Data Retention

In this section, we show how error profiling impacts end-to-end reliability. We study the bit error rate of a system that uses a bit-granularity repair mechanism (e.g., such as those discussed in §2.2) to reduce the DRAM refresh rate, which prior work shows can significantly improve overall system performance and energy-efficiency [111, 147, 150, 171] and enable continued density scaling [136]. We assume that data-retention errors follow the error model described in §2.4 (i.e., uniformly with a fixed raw bit error rate, which is consistent with prior experimental studies [12, 47, 85, 146, 147, 161, 166]) and that the repair mechanism perfectly repairs any at-risk bits that are identified by either active or reactive profiling. We assume a (71, 64) SEC on-die ECC code and a secondary ECC capable of detecting and correcting a single error in each on-die ECC word during reactive profiling.

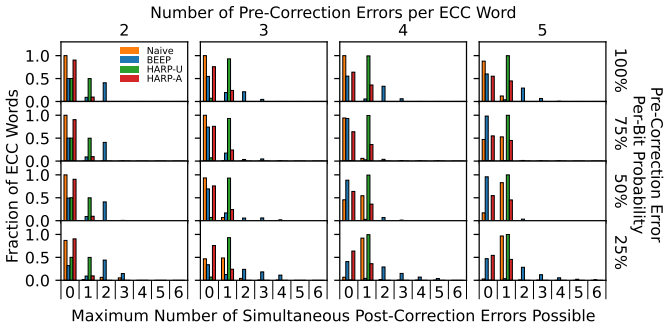
Fig. 10 illustrates the fraction of all bits that are at risk of post-correction errors (i.e., the bit error rate) before (Fig. 10, left) and after (Fig. 10, right) secondary ECC is applied (i.e., before and after performing reactive profiling) given an x-axis number of active profiling rounds. Each line marker shows a different data-retention RBER (e.g., due to operating at different refresh rates).

We make three observations. First, all profilers in Fig. 10 (left) behave consistently with the coverage analysis of §7.2.1. HARP quickly identifies all bits at risk of direct error, but leaves indirect errors to be identified by reactive profiling. Both Naive and BEEP slowly explore different combinations of pre-correction errors, with Naive steadily reducing the BER given more profiling rounds while BEEP fails to do so.

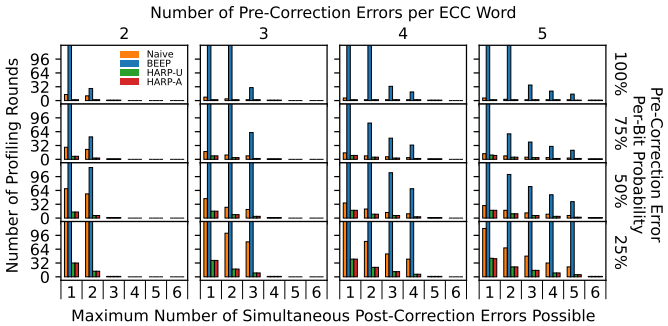
Second, Fig. 10 (left) shows the benefit of HARP-A knowing the on-die ECC function. While both HARP-U and HARP-A quickly identify all bits at risk of direct errors, HARP-A also identifies bits at risk of indirect errors, thereby considerably reducing the overall BER (and therefore, the total number of bits) that remain to be identified by reactive profiling.

Third, Fig. 10 (right) shows that both HARP[12] and Naive reach a BER of zero after sufficiently many profiling rounds, though Naive

[12]HARP-A is not shown in Fig. 10 (right) because it exhibits *identical* BER to HARP-U after applying secondary ECC (i.e., because both profilers have identical coverage of bits at risk of direct errors).



(a) Normalized histogram of the maximum number of simultaneous post-correction errors (x-axis) possible across all simulated ECC words after 128 rounds of profiling.



(b) Number of profiling rounds (y-axis) required to achieve 99th-percentile values of the maximum number of simultaneous post-correction errors possible (x-axis).

Figure 9: Maximum number of simultaneous post-correction errors possible given all at-risk bits missed after 128 rounds of profiling.

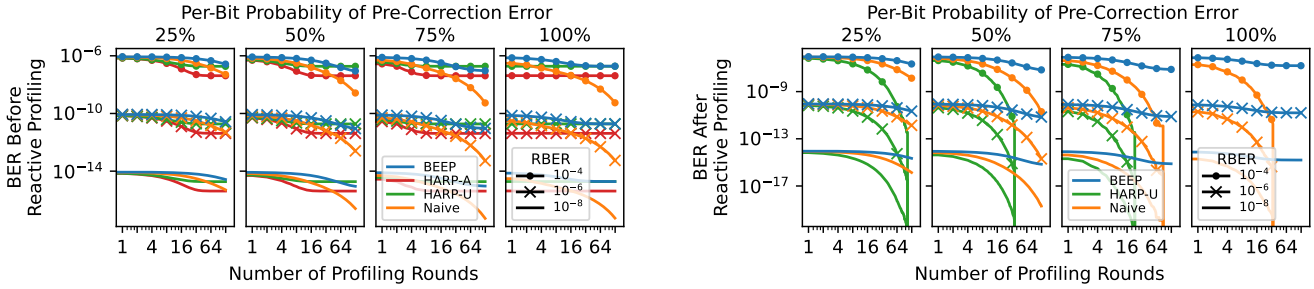


Figure 10: Data-retention bit error rate (BER) using an ideal repair mechanism before (left) and after (right) applying the secondary ECC.

takes significantly more profiling rounds to do so (e.g., 3.7× for a per-bit pre-correction error probability of 75%). This behavior is consistent with the fact that both profilers eventually achieve full coverage of bits at risk of direct error (shown in §7.2.1) In contrast, BEEP fails to reach a zero probability value because it fails to achieve full coverage of bits at risk of direct error. Note that HARP-U immediately identifies all bits at risk of direct errors in the first profiling round with a per-bit pre-correction error probability of 100%, so it is not visible in the rightmost plot.

We conclude that HARP effectively identifies all bits at-risk of error faster than the baseline profilers, thereby enabling the repair mechanism to safely operate at the evaluated raw bit error rates. Although this case study uses a simple data-retention error model that does not include low-probability errors or other failure modes (discussed in §2.4), it demonstrates (1) the importance of a practical and effective error profiling algorithm in enabling a repair mechanism to mitigate errors; and (2) the advantages that HARP provides in an end-to-end setting by overcoming the error profiling challenges introduced by on-die ECC.

8 RELATED WORK

To our knowledge, this is the first work to (i) conduct an analytical study of how system-level error profiling interacts with on-die ECC, and (ii) propose a bit-granularity error profiling algorithm to support memory chips with on-die ECC. Main memory error profiling is a long-standing and difficult problem that prior works have tackled in many different ways. In our work, we focus on error profiling algorithms in the context of on-die ECC. We briefly review related works which we already compared to in prior sections.

Profiling Without On-die ECC. Prior works propose various error profiling algorithms in the context of DRAM [11, 12, 24, 27, 29, 30, 40, 76–79, 82, 84, 86, 87, 103, 104, 109–111, 132, 145, 147, 150, 159, 171] and emerging main memory technologies such as PCM and STT-RAM [48, 149, 168, 183]. To our knowledge, none of these works identify or address the effects that on-die ECC has on error profiling. Furthermore, many of the insights and/or solutions developed by these works (e.g., algorithms for identifying low-probability errors [77, 147, 149, 150, 159]) are complementary to our work and can be integrated with HARP (e.g., during active profiling) to improve error coverage.

Profiling With On-die ECC. To our knowledge, only two works [145, 146] study how on-die ECC impacts memory error characterization and profiling in detail. Unfortunately, neither work identifies the challenges that on-die ECC introduces for error profiling. Of these works, only BEEP [145] is a profiling algorithm that accounts for on-die ECC. However, BEEP focuses on reverse-engineering *pre-correction* error locations using a slow algorithm that we show is not well suited for identifying bits at risk of *post-correction* error and still suffers from the three profiling challenges. In contrast, we comprehensively study the three key challenges that on-die ECC introduces for bit-granularity error profiling (§4), which we address by proposing and evaluating HARP.

9 CONCLUSION

We study how on-die ECC affects memory error profiling and identify three key challenges that it introduces: on-die ECC (1) exponentially increases the number of at-risk bits the profiler must identify; (2) makes individual at-risk bits more difficult to identify; and (3) interferes with commonly-used memory data patterns. To overcome these three challenges, we introduce Hybrid Active-Reactive Profiling (HARP), a new bit-granularity error profiling algorithm that enables practical and effective error profiling for memory chips that use on-die ECC. HARP exploits the key idea that on-die ECC introduces two different sources of post-correction errors: (1) direct errors that result from pre-correction errors within the data portion of the ECC codeword; and (2) indirect errors that are a result of the on-die ECC correction process. If all bits at risk of direct error are identified, the number of concurrent indirect errors is upper-bounded by the correction capability of on-die ECC. Therefore, HARP uses simple modifications to the on-die ECC mechanism to quickly identify bits at risk of direct errors and relies on a secondary ECC within the memory controller to safely identify indirect errors. Our evaluations show that HARP achieves full coverage of all at-risk bits in memory chips that use on-die ECC faster than prior approaches to error profiling. We hope that the studies, analyses, and ideas we provide in this work will enable researchers and practitioners alike to think about and overcome the challenge of how to handle error detection and correction across the hardware-software stack in the presence of on-die ECC.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of MICRO 2021 for their feedback and the SAFARI Research Group members for their feedback and the stimulating intellectual environment they provide. We acknowledge the generous gift funding provided by our industrial partners: Google, Huawei, Intel, Microsoft, and VMware.

REFERENCES

[1] BEER Source Code. https://github.com/CMU-SAFARI/BEER.
[2] EINSim Source Code. https://github.com/CMU-SAFARI/EINSim.
[3] HARP Source Code. https://github.com/CMU-SAFARI/HARP.
[4] R Dean Adams. *High Performance Memory Testing: Design Principles, Fault Modeling and Self-Test.* Springer SBM. 2002.
[5] Zaid Al-Ars. *DRAM Fault Analysis and Test Generation.* Ph.D. Dissertation. TU Delft. 2005.
[6] Alaa R Alameldeen, Ilya Wagner, Zeshan Chishti, Wei Wu, Chris Wilkerson, and Shih-Lien Lu. Energy-Efficient Cache Design Using Variable-Strength Error-Correcting Codes. *ISCA*. 2011.
[7] Alliance Memory. *2Gb/4Gb/8Gb LPDDR4.* Alliance Memory. 2020. Rev. 1.0.
[8] AMD. BKDG for AMD NPT Family 0Fh Processors. http://developer.amd.com/wordpress/media/2012/10/325591.pdf. 2009.
[9] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, et al. Spin-Transfer Torque Magnetic Random Access Memory (STT-MRAM). *JETC*. 2013.
[10] Manu Awasthi, Manjunath Shevgoor, Kshitij Sudan, Bipin Rajendran, Rajeev Balasubramonian, and Viii Srinivasan. Efficient Scrub Mechanisms for Error-Prone Emerging Memories. In *HPCA*. 2012.
[11] Angelo Bacchini, Marco Rovatti, Gianluca Furano, and Marco Ottavi. Characterization of Data Retention Faults in DRAM Devices. In *DFT*. 2014.
[12] Seungjae Baek, Sangyeun Cho, and Rami Melhem. Refresh Now and Then. In *TC*. 2014.
[13] Kuljit S. Bains, Rajat Agarwal, and Jongwon Lee. Read Retry To Selectively Disable On-Die ECC. US Patent Application 20,200,278,906. 2020.
[14] Michael A Bajura, Younes Boulghassoul, Riaz Naseer, Sandeepan DasGupta, Arthur F Witulski, Jeff Sondeen, et al. Models and Algorithmic Limits for an ECC-Based Approach To Hardening Sub-100-nm SRAMs. *Trans. on Nucl. Sci.* 2007.
[15] Leonardo Bautista-Gomez, Ferad Zyulkyarov, Osman Unsal, and Simon McIntosh-Smith. Unprotected Computing: A Large-Scale Study of DRAM Raw Error Rate on a Supercomputer. In *SC*. 2016.
[16] Raj Chandra Bose and Dwijendra K Ray-Chaudhuri. On a Class of Error Correcting Binary Group Codes. *Information and Control*. 1960.
[17] Geoffrey W Burr, Matthew J Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, et al. Phase Change Memory Technology. *J Vac Sci Technol B*. 2010.
[18] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Error Characterization, Mitigation, and Recovery In Flash-Memory-Based Solid-State Drives. *Proc. IEEE*. 2017.
[19] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery. *Inside Solid State Drives*. 2018.
[20] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis. In *DATE*. 2012.
[21] Sanguhn Cha, O Seongil, Hyunsung Shin, Sangjoon Hwang, Kwangil Park, Seong Jin Jang, et al. Defect Analysis and Cost-Effective Resilience Architecture for Future DRAM Devices. In *HPCA*. 2017.
[22] Karthik Chandrasekar, Sven Goossens, Christian Weis, Martijn Koedam, Benny Akesson, Norbert Wehn, et al. Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization. In *DATE*. 2014.
[23] Kevin K Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, et al. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *SIGMETRICS*. 2016.
[24] Kevin K Chang, A Giray Yaălikçi, Saugata Ghose, Aditya Agrawal, Niladrish Chatterjee, Abhijith Kashyap, et al. Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms. In *SIGMETRICS*. 2017.
[25] E Chen, D Apalkov, Z Diao, A Driskill-Smith, D Druist, D Lottis, et al. Advances and Future Prospects of Spin-Transfer Torque Random Access Memory. *TOM*. 2010.
[26] Kuo-Liang Cheng, Ming-Fu Tsai, and Cheng-Wen Wu. Neighborhood Pattern-Sensitive Fault Testing and Diagnostics for Random-Access Memories. *TCADICS*. 2002.
[27] Haerang Choi, Dosun Hong, Jaesung Lee, and Sungjoo Yoo. Reducing DRAM Refresh Power Consumption by Runtime Profiling of Retention Time and Dual-Row Activation. *Microprocessors and Microsystems*. 2020.
[28] Ki Chul Chun, Hui Zhao, Jonathan D Harms, Tae-Hyoung Kim, Jian-Ping Wang, and Chris H Kim. A Scaling Roadmap and Performance Evaluation of In-Plane and Perpendicular MTJ Based STT-MRAMs for High-Density Cache Memory. *JSSC*. 2012.
[29] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, et al. Are We Susceptible to RowHammer? An End-to-End Methodology for Cloud Providers. In *S&P*. 2020.
[30] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against RowHammer Attacks. In *S&P*. 2019.
[31] Daniel J Costello and Shu Lin. *Error Control Coding: Fundamentals and Applications.* Prentice Hall. 1982.
[32] Kjersten Criss, Kuljit Bains, Rajat Agarwal, Tanj Bennett, Terry Grunzke, Jangryul Keith Kim, et al. Improving Memory Reliability by Bounding DRAM Faults: DDR5 Improved Reliability Features. In *MEMSYS*. 2020.
[33] Xiaole Cui, Zuolin Cheng, Chunglen Lee, Xinnan Lin, Yiqun Wei, Xiaogang Chen, et al. A Snake Addressing Scheme for Phase Change Memory Testing. *SCIS*. 2016.
[34] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R Hanebutte, and Onur Mutlu. Memory Power Management via Dynamic Voltage/Frequency Scaling. In *ICAC*. 2011.
[35] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*. 2008.
[36] Rob Dekker, Frans Beenker, and Loek Thijssen. A Realistic Fault Model and Test Algorithms for Static Random Access Memories. *TCAD*. 1990.
[37] Timothy J Dell. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. *IBM Microelectronics Division*. 1997.
[38] Robert H Dennard. Field-Effect Transistor Memory. US Patent 3,387,286. 1968.
[39] Everspin Technologies. 16Mb MRAM MR4A16B. https://www.everspin.com/16mb-mram-parallel-interface. 2021.
[40] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, et al. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *IEEE S&P*. 2020.
[41] Spencer M Gold and Arun B Hegde. Providing Test Coverage of Integrated ECC Logic en Embedded Memory. US Patent 8,914,687. 2014.
[42] Seong-Lyong Gong, Jungrae Kim, and Mattan Erez. DRAM Scaling Error Evaluation Model Using Various Retention Time. In *DSN-W*. 2017.
[43] Seong-Lyong Gong, Jungrae Kim, Sangkug Lym, Michael Sullivan, Howard David, and Mattan Erez. DUO: Exposing On-Chip Redundancy to Rank-Level ECC for High Reliability. In *HPCA*. 2018.
[44] Kevin W Gorman, Michael R Ouellette, and Patrick E Perry. Memory Test With In-Line Error Correction Code Logic. US Patent 9,224,503. 2015.
[45] Xiaochen Guo, Mahdi Nazm Bojnordi, Qing Guo, and Engin Ipek. Sanitizer: Mitigating the Impact of Expensive ECC Checks on STT-MRAM Based Main Memories. *TOC*. 2017.
[46] T Hamamoto, S Sugiura, and S Sawada. Well Concentration: A Novel Scaling Limitation Factor Derived From DRAM Retention Time and Its Modeling. In *IEDM*. 1995.
[47] Takeshi Hamamoto, Soichi Sugiura, and Shizuo Sawada. On the Retention Time Distribution of Dynamic Random Access Memory (DRAM). In *TED*. 1998.
[48] Said Hamdioui, Peyman Pouyan, Huawei Li, Ying Wang, Arijit Raychowdhur, and Insik Yoon. Test and Reliability of Emerging Non-Volatile Memories. In *ATS*. 2017.
[49] Richard W Hamming. Error Detecting and Error Correcting Codes. In *Bell Labs Technical Journal*. 1950.
[50] Yinhe Han, Ying Wang, Huawei Li, and Xiaowei Li. Data-Aware DRAM Refresh to Squeeze the Margin of Retention Time in Hybrid Memory Cube. In *ICCAD*. 2014.
[51] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, et al. Array Programming with NumPy. *Nature*. 2020.
[52] Hasan Hassan, Yahya C. Tugrul, Jeremie S. Kim, Victor Van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. In *MICRO*. 2021.
[53] Alexis Hocquenghem. Codes Correcteurs D'erreurs. *Chiffres*. 1959.
[54] Sungjoo Hong. Memory Technology Trend and Future Challenges. In *IEDM*. 2010.
[55] Masashi Horiguchi and Kiyoo Itoh. *Nanoscale Memory Repair.* Springer SBM. 2011.
[56] Yiming Huai et al. Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects. *AAPPS Bulletin*. 2008.
[57] J. D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*. 2007.
[58] 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.
[59] Intelligent Memory. I'M ECC DRAM with Integrated Error Correcting Code. Product Brief. 2016.
[60] Intelligent Memory. I'M ECC DRAM with Integrated Error Correcting Code. Product Brief. 2020.
[61] Engin Ipek, Jeremy Condit, Edmund B Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically Replicated Memory: Building Reliable Systems from Nanoscale Resistive Memories. In *ASPLOS*. 2010.
[62] ITRS. More Moore. 2015. www.itrs2.net.

[63] Bruce Jacob, David Wang, and Spencer Ng. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann. 2010. Chapter 30.3: "Memory Errors and Error Correction".
[64] JEDEC. *DDR4 SDRAM Specification*. 2012.
[65] JEDEC. Low Power Double Data Rate 4 (LPDDR4) SDRAM Specification. *JEDEC Standard JESD209–4B*. 2014.
[66] JEDEC. *JEP122H: Failure Mechanisms and Models for Semiconductor Devices*. 2016.
[67] JEDEC. *DDR5 SDRAM Specification*. 2020.
[68] JEDEC. Main Memory: DDR4 & DDR5 SDRAM. https://www.jedec.org/category/technology-focus-area/main-memory-ddr3-ddr4-sdram. 2021.
[69] Sangmok Jeong, SeungYup Kang, and Joon-Sung Yang. PAIR: Pin-Aligned In-DRAM ECC Architecture Using Expandability of Reed-Solomon Code. In *DAC*. 2020.
[70] Seonghoon Jin, Jeong-Hyong Yi, Jae Hoon Choi, Dae Gwan Kang, Young June Park, and Hong Shick Min. Prediction of Data Retention Time Distribution of DRAM by Physics-Based Statistical Simulation. *TED*. 2005.
[71] Matthias Jung, Deepak M Mathew, Carl C Rheinländer, Christian Weis, and Norbert Wehn. A Platform to Analyze DDR3 DRAM's Power and Retention Time. *IEEE Design & Test*. 2017.
[72] Matthias Jung, Deepak M Mathew, Christian Weis, and Norbert Wehn. Approximate Computing With Partially Unreliable Dynamic Random Access Memory-Approximate DRAM. In *DAC*. 2016.
[73] Sangbeom Kang, Woo Yeong Cho, Beak-Hyung Cho, Kwang-Jin Lee, Chang-Soo Lee, Hyung-Rok Oh, et al. A 0.1-um 1.8-V 256-Mb Phase-Change Random Access Memory (PRAM) With 66-MHz Synchronous Burst-Read Operation. *JSSC*. 2006.
[74] Uksong Kang, Hak-soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, et al. Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling. In *The Memory Forum*. 2014.
[75] Mohammad Nasim Imtiaz Khan and Swaroop Ghosh. Test Challenges and Solutions for Emerging Non-Volatile Memories. In *VTS*. 2018.
[76] Samira Khan, Donghyuk Lee, Yoongu Kim, Alaa R Alameldeen, Chris Wilkerson, and Onur Mutlu. The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study. In *SIGMETRICS*. 2014.
[77] Samira Khan, Donghyuk Lee, and Onur Mutlu. PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM. In *DSN*. 2016.
[78] Samira Khan, Chris Wilkerson, Donghyuk Lee, Alaa R Alameldeen, and Onur Mutlu. A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM. In *CAL*. 2016.
[79] Samira Khan, Chris Wilkerson, Zhe Wang, Alaa R Alameldeen, Donghyuk Lee, and Onur Mutlu. Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content. In *MICRO*. 2017.
[80] Dong Wan Kim and Mattan Erez. Balancing Reliability, Cost, and Performance Tradeoffs With FreeFault. In *HPCA*. 2015.
[81] Dong Wan Kim and Mattan Erez. RelaxFault Memory Repair. In *ISCA*. 2016.
[82] Joohee Kim and Marios C Papaefthymiou. Block-Based Multiperiod Dynamic Memory Design for Low Data-Retention Power. In *TVLSI*. 2003.
[83] Jungrae Kim, Michael Sullivan, and Mattan Erez. Bamboo ECC: Strong, Safe, and Flexible Codes For Reliable Computer Memory. In *HPCA*. 2015.
[84] Jeremie S Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines. In *ICCD*. 2018.
[85] Jeremie S. Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices. In *HPCA*. 2018.
[86] Jeremie S Kim, Minesh Patel, Hasan Hassan, Lois Orosa, and Onur Mutlu. D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers With Low Latency And High Throughput. In *HPCA*. 2019.
[87] Jeremie S. Kim, Minesh Patel, Abdullah Giray Yaglikci, Hasan Hassan, Roknoddin Azizi, Lois Orosa, et al. Revisiting RowHammer: An Experimental Analysis of Modern Devices and Mitigation Techniques. In *ISCA*. 2020.
[88] Kinarn Kim and Su Jin Ahn. Reliability Investigations for Manufacturable High Density PRAM. In *IRPS*. 2005.
[89] Kinam Kim, Chang-Gyu Hwang, and Jong Gil Lee. DRAM Technology Perspective for Gigabit Era. *TED*. 1998.
[90] Kinam Kim and Jooyoung Lee. A New Investigation of Data Retention Time in Truly Nanoscaled DRAMs. In *EDL*. 2009.
[91] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, et al. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*. 2014.
[92] Donald Kline, Rami Melhem, and Alex K Jones. Sustainable Fault Management and Error Correction for Next-Generation Main Memories. In *IGSC*. 2017.
[93] Donald Kline, Jiangwei Zhang, Rami Melhem, and Alex K Jones. Flower and Fame: A Low Overhead Bit-Level Fault-Map and Fault-Tolerance Approach for Deeply Scaled Memories. In *HPCA*. 2020.
[94] Wei Kong, Paul C Parries, G Wang, and Subramanian S Iyer. Analysis of Retention Time Distribution of Embedded DRAM-A New Method to Characterize Across-Chip Threshold Voltage Variation. In *ITC*. 2008.
[95] Skanda Koppula, Lois Orosa, A Giray Yağlıkçı, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, et al. EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM. In *MICRO*. 2019.
[96] Kira Kraft, Chirag Sudarshan, Deepak M Mathew, Christian Weis, Norbert Wehn, and Matthias Jung. Improving the Error Behavior of DRAM by Exploiting its Z-Channel Property. In *DATE*. 2018.
[97] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. In *ISPASS*. 2013.
[98] Nohhyup Kwak, Saeng-Hwan Kim, Kyong Ha Lee, Chang-Ki Baek, Mun Seon Jang, Yongsuk Joo, et al. A 4.8 Gb/s/pin 2Gb LPDDR4 SDRAM with Sub-100μA Self-Refresh Current for IoT Applications. In *ISSCC*. 2017.
[99] Hye-Jung Kwon, Eunsung Seo, Chan-Yong Lee, Young-Hun Seo, Gong-Heum Han, Hye-Ran Kim, et al. An Extremely Low-Standby-Power 3.733 Gb/s/pin 2Gb LPDDR4 SDRAM for Wearable Devices. In *ISSCC*. 2017.
[100] Sanghyuk Kwon, Young Hoon Son, and Jung Ho Ahn. Understanding DDR4 in Pursuit of In-DRAM ECC. In *ISOCC*. 2014.
[101] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA*. 2009.
[102] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, et al. Phase-Change Technology and the Future of Main Memory. *IEEE Micro*. 2010.
[103] Donghyuk Lee, Samira Khan, Lavanya Subramanian, Saugata Ghose, Rachata Ausavarungnirun, Gennady Pekhimenko, et al. Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms. In *SIGMETRICS*. 2017.
[104] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, et al. Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case. In *HPCA*. 2015.
[105] Suyoun Lee, Jeung-hyun Jeong, Taek Sung Lee, Won Mok Kim, and Byung-ki Cheong. A Study on the Failure Mechanism of a Phase-Change Memory in Write/Erase Cycling. *EDL*. 2009.
[106] Seunghak Lee, Nam Sung Kim, and Daehoon Kim. Exploiting OS-Level Memory Offlining for DRAM Power Management. *CAL*. 2019.
[107] Seok-Hee Lee. Technology Scaling Challenges and Opportunities of Memory Devices. In *IEDM*. 2016.
[108] Udo Lieneweg, D Nguyen, and B Blaes. Assesment of DRAM Reliability from Retention Time Measurements. *Flight Readiness Technol. Assessment NASA EEE Parts Prog*. 1998.
[109] Chung Hsiang Lin, De-Yu Shen, Yi-Jung Chen, Chia-Lin Yang, and Michael Wang. SECRET: Selective Error Correction for Refresh Energy Reduction in DRAMs. In *ICCD*. 2012.
[110] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms. In *ISCA*. 2013.
[111] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*. 2012.
[112] Stephen Longofono, Donald Kline Jr, Rami Melhem, and Alex K Jones. Predicting and Mitigating Single-Event Upsets in DRAM Using HOTH. *Microelectronics Reliability*. 2021.
[113] Yixin Luo, Saugata Ghose, Yu Cai, Erich F Haratsch, and Onur Mutlu. Heat-Watch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness. In *HPCA*. 2018.
[114] Yixin Luo, Saugata Ghose, Yu Cai, Erich F Haratsch, and Onur Mutlu. Improving 3D NAND Flash Memory Lifetime by Tolerating Early Retention Loss and Process Variation. In *SIGMETRICS*. 2018.
[115] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, et al. Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory. In *DSN*. 2014.
[116] Jack A Mandelman, Robert H Dennard, Gary B Bronner, John K DeBrosse, Rama Divakaruni, Yujun Li, et al. Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM). In *IBM JRD*. 2002.
[117] Timothy C May and Murray H Woods. Alpha-Particle-Induced Soft Errors in Dynamic Memories. *TED*. 1979.
[118] mcelog. *Bad Page Offlining*. mcelog. 2021. https://mcelog.org/badpageofflining.html.
[119] J Meza et al. A Large-Scale Study of Flash Memory Errors in the Field. In *SIGMETRICS*. 2015.
[120] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. In *DSN*. 2015.
[121] Micron Technology Inc. 2017. *ECC Brings Reliability and Power Efficiency to Mobile Devices*. Technical Report. Micron Technology Inc.
[122] Micron Technology Inc. 2020. *TN-40-40: DDR4 Point-to-Point Design Guide*. Technical Report. Micron Technology Inc.
[123] Todd K Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. John Wiley & Sons. 2005.

[124] Y. Mori, K. Ohyu, K. Okonogi, and R. i. Yamada. The Origin of Variable Retention Time in DRAM. In *IEDM*. 2005.

[125] Ireneusz Mrozek. Analysis of Multibackground Memory Testing Techniques. 2010.

[126] Ireneusz Mrozek. *Multi-Run Memory Tests for Pattern Sensitive Faults*. Springer. 2019.

[127] Lev Mukhanov, Dimitrios S Nikolopoulos, and Georgios Karakonstantis. DStress: Automatic Synthesis of DRAM Reliability Stress Viruses Using Genetic Algorithms. In *MICRO*. 2020.

[128] Shubhendu S Mukherjee, Joel Emer, Tryggve Fossum, and Steven K Reinhardt. Cache Scrubbing in Microprocessors: Myth or Necessity?. In *SDC*. 2004.

[129] Onur Mutlu. Memory Scaling: A Systems Architecture Perspective. In *IMW*. 2013.

[130] Onur Mutlu. Main Memory Scaling: Challenges and Solution Directions. In *More than Moore Technologies for Next Generation Computer Design*. Springer, 127–153. 2015.

[131] Onur Mutlu. The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser. In *DATE*. 2017.

[132] Onur Mutlu and Jeremie Kim. RowHammer: A Retrospective. In *TCAD*. 2019.

[133] Onur Mutlu and Lavanya Subramanian. Research Problems and Opportunities in Memory Systems. In *SUPERFRI*. 2014.

[134] Helia Naeimi, Charles Augustine, Arijit Raychowdhury, Shih-Lien Lu, and James Tschanz. STTRAM Scaling and Retention Failure. *Intel Technology Journal*. 2013.

[135] Prashant J Nair, Bahar Asgari, and Moinuddin K Qureshi. SuDoku: Tolerating High-Rate of Transient Failures for Enabling Scalable STTRAM. In *DSN*. 2019.

[136] Prashant J Nair, Dae-Hyun Kim, and Moinuddin K Qureshi. ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates. In *ISCA*. 2013.

[137] Prashant J Nair, Vilas Sridharan, and Moinuddin K Qureshi. XED: Exposing On-Die Error Detection Information for Strong Memory Reliability. In *ISCA*. 2016.

[138] Omar Naji, Christian Weis, Matthias Jung, Norbert Wehn, and Andreas Hansson. A High-Level DRAM Timing, Power And Area Exploration Tool. In *SAMOS*. 2015.

[139] NVIDIA. *Dynamic Page Retirement*. NVIDIA. 2020. https://docs.nvidia.com/deploy/dynamic-page-retirement/index.html.

[140] Tae-Young Oh, Hoeju Chung, Jun-Young Park, Ki-Won Lee, Seunghoon Oh, Su-Yeon Doo, et al. A 3.2 Gbps/pin 8 Gbit 1.0 V LPDDR4 SDRAM with Integrated ECC Engine for Sub-1 V DRAM Core Operation. *JSSC*. 2014.

[141] Lois Orosa, Abdullah Giray Yağlıkçı, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, et al. A Deeper Look into RowHammer's Sensitivities: Experiemental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses. In *MICRO*. 2021.

[142] Sung-Il Pae, Vivek Kozhikkottu, Dinesh Somasekar, Wei Wu, Shankar Ganesh Ramasubramanian, Melin Dadual, et al. Minimal Aliasing Single-Error-Correction Codes for DRAM Reliability Improvement. *IEEE Access*. 2021.

[143] Kyungbae Park, Donghyuk Yun, and Sanghyeon Baeg. Statistical Distributions of Row-Hammering Induced Failures in DDR3 Components. *Microelectronics Reliability*. 2016.

[144] Sung-Kye Park. Technology Scaling Challenge and Future Prospects of DRAM and NAND Flash Memory. In *IMW*. 2015.

[145] Minesh Patel, Jeremie Kim, Taha Shahroodi, Hasan Hassan, and Onur Mutlu. Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics. In *MICRO*. 2020.

[146] Minesh Patel, Jeremie S Kim, Hasan Hassan, and Onur Mutlu. Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices. In *DSN*. 2019.

[147] Minesh Patel, Jeremie S Kim, and Onur Mutlu. The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions. In *ISCA*. 2017.

[148] Minesh Patel, Geraldo F. Oliveira, and Onur Mutlu. HARP Artifacts. *ZENODO*. 2021. doi:10.5281/zenodo.5148592.

[149] Moinuddin K Qureshi. Pay-As-You-Go: Low-Overhead Hard-Error Correction for Phase Change Memories. In *MICRO*. 2011.

[150] Moinuddin K Qureshi, Dae-Hyun Kim, Samira Khan, Prashant J Nair, and Onur Mutlu. AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems. In *DSN*. 2015.

[151] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *ISCA*. 2009.

[152] Arijit Raychowdhury, Dinesh Somasekhar, Tanay Karnik, and Vivek De. Design Space and Scalability Exploration of 1T-1STT MTJ Memory Arrays in the Presence of Variability and Disturbances. In *IEDM*. 2009.

[153] Phillip J Restle, JW Park, and Brian F Lloyd. DRAM Variable Retention Time. In *IEDM*. 1992.

[154] Tom Richardson and Ruediger Urbanke. *Modern Coding Theory*. Cambridge University Press. 2008.

[155] Ron M Roth. *Introduction to Coding Theory*. Cambridge University Press. 2006.

[156] Abdallah M Saleh, Juan J Serrano, and Janak H Patel. Reliability of Scrubbing Recovery-Techniques for Memory Systems. *TR*. 1990.

[157] Stuart Schechter, Gabriel H Loh, Karin Strauss, and Doug Burger. Use ECP, Not ECC, for Hard Failures in Resistive Memories. *ISCA*. 2010.

[158] Nak Hee Seong, Dong Hyuk Woo, Vijayalakshmi Srinivasan, Jude A Rivers, and Hsien-Hsin S Lee. SAFER: Stuck-at-Fault Error Recovery for Memories. In *MICRO*. 2010.

[159] Rasool Sharifi and Zainalabedin Navabi. Online Profiling for Cluster-Specific Variable Rate Refreshing in High-Density DRAM Systems. In *ETS*. 2017.

[160] Wongyu Shin, Jungwhan Choi, Jaemin Jang, Jinwoong Suh, Youngsuk Moon, Yongkee Kwon, et al. DRAM-Latency Optimization Inspired by Relationship Between Row-Access Time and Refresh Timing. *TOC*. 2015.

[161] C Glenn Shirley and W Robert Daasch. Copula Models of Correlation: A DRAM Case Study. In *TC*. 2014.

[162] Young Hoon Son, Sukhan Lee, O Seongil, Sanghyuk Kwon, Nam Sung Kim, and Jung Ho Ahn. CiDRA: A Cache-Inspired DRAM Resilience Architecture. In *HPCA*. 2015.

[163] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B Ferreira, Jon Stearley, John Shalf, et al. Memory Errors in Modern Systems: The Good, the Bad, and the Ugly. In *ASPLOS*. 2015.

[164] Vilas Sridharan and Dean Liberty. A Study of DRAM Failures in the Field. In *SC*. 2012.

[165] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults. In *SC*. 2013.

[166] Soubhagya Sutar, Arnab Raha, and Vijay Raghunathan. D-PUF: An Intrinsically Reconfigurable DRAM PUF for Device Authentication in Embedded Systems. In *CASES*. 2016.

[167] Synopsys. 2015. *Reliability, Availability and Serviceability (RAS) for Memory Interfaces*. Technical Report. Synopsys.

[168] Mohammad Khavari Tavana, Amir Kavyan Ziabari, Mohammad Arjomand, Mahmut Kandemir, Chita Das, and David Kaeli. REMAP: A Reliability/Endurance Mechanism for Advancing PCM. In *MEMSYS*. 2017.

[169] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace. 2009.

[170] Elena Ioana Vatajelu, Peyman Pouyan, and Said Hamdioui. State of the Art and Challenges for Test and Reliability of Emerging Nonvolatile Resistive Memories. *JCTA*. 2018.

[171] Ravi K Venkatesan, Stephen Herr, and Eric Rotenberg. Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM. In *HPCA*. 2006.

[172] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, et al. SciPy 1.0: Fundamental Algorithms For Scientific Computing In Python. *Nature Methods*. 2020.

[173] Matt Von Thun. Qualification and Reliability of MRAM Toggle Memory Designed for Space Applications. https://www.everspin.com/file/157395/download. 2020.

[174] Hao Wang. *Architecting Memory Systems Upon Highly Scaled Error-Prone Memory Technologies*. Ph.D. Dissertation. Rensselaer Polytechnic Institute. 2017.

[175] Christian Weis, Matthias Jung, Peter Ehses, Cristiano Santos, Pascal Vivet, Sven Goossens, et al. Retention Time Measurements and Modelling of Bit Error Rates of Wide I/O DRAM in MPSoCs. In *DATE*. 2015.

[176] Chris Wilkerson, Hongliang Gao, Alaa R Alameldeen, Zeshan Chishti, Muhammad Khellah, and Shih-Lien Lu. Trading Off Cache Capacity for Reliability to Enable Low Voltage Operation. *ISCA*. 2008.

[177] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, et al. Phase Change Memory. *Proc. IEEE*. 2010.

[178] David S Yaney, Chih-Yuan Lu, Ross A Kohler, Michael J Kelly, and James T Nelson. A Meta-Stable Leakage Phenomenon in DRAM Charge Storage-Variable Hold Time. In *IEDM*. 1987.

[179] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P Jouppi, and Mattan Erez. FREE-p: Protecting Non-Volatile Memory Against Both Hard and Soft Errors. In *HPCA*. 2011.

[180] Jiangwei Zhang, Donald Kline, Liang Fang, Rami Melhem, and Alex K Jones. Dynamic Partitioning To Mitigate Stuck-At Faults in Emerging Memories. In *ICCAD*. 2017.

[181] Xinmiao Zhang. In *VLSI Architectures for Modern Error-Correcting Codes*. 53. 2015.

[182] Xianwei Zhang, Youtao Zhang, Bruce R Childers, and Jun Yang. Restore Truncation for Performance Improvement in Future DRAM Systems. In *HPCA*. 2016.

[183] Zhe Zhang, Weijun Xiao, Nohhyun Park, and David J Lilja. Memory Module-Level Testing and Error Behaviors for Phase Change Memory. In *ICCD*. 2012.

[184] Bo Zhao. *Improving Phase Change Memory (PCM) and Spin-Torque-Transfer Magnetic-RAM (STT-MRAM) as Next-Generation Memories: A Circuit Perspective*. Ph.D. Dissertation. University of Pittsburgh. 2014.

A ARTIFACT DESCRIPTION APPENDIX

A.1 Abstract

Our artifacts provide the source code needed to replicate all experiments in the paper, including all figures. The artifacts comprise two parts: (1) Monte-Carlo simulations that generate raw data; and (2) data analysis scripts to parse, aggregate, and plot the data. This appendix describes both parts and how to run them to replicate our experiments.

A.2 Artifact Check-list (Meta-information)

Parameter	Value
Program	C++ program and Python3/shell scripts
Compilation	C++11 compiler (tested with GCC 8)
Run-time environment	Debian 10 (or similar) Linux
Hardware	Standard linux system (>= 40 GB RAM recommended for the analysis scripts)
Output	Plaintext data files and PDF files containing plots
Experiment workflow	Run simulations, aggregate output files, and run analysis scripts on the outputs
Experiment customization	Number of simulation jobs (i.e., Monte-Carlo samples)
Disk space requirement	≈10-100 GB
Workflow preparation time	≈1 day
Experiment completion time	≈1-2 weeks
Publicly available?	Zenodo [148] and GitHub [3]
Code licenses	MIT

A.3 Description

A.3.1 How to Access. The artifacts are available on Zenodo with DOI 10.5281/zenodo.5148592 [148] and on GitHub [3].

A.3.2 Hardware Dependencies. The artifacts are designed to run on a typical Linux system. As §A.7 discusses in detail, the C++ simulations are highly parallelziable with low memory requirements, so a system with more CPUs will reduce the overall simulation time required. In contrast, the analysis scripts run as single tasks that consume memory in proportion to the amount of simulation data (up to 40 GB of memory for the full evaluation configuration described in §A.7). Therefore a single machine with a large memory may be necessary.

A.3.3 Software Dependencies.

- GNU Make
- C++11 build toolchain (tested with GCC 8 on Debian 10)
- Python 3 [169] with `matplotlib` [57], `scipy` [172], `numpy` [51]

A.4 Installation

No system installation is required; the C++ code can all be built and run in place. The C++ application depends on both Z3 [35] and EINSim [2], and we have: (1) included copies of both source distributions in the Zenodo distribution for sake of convenience; and (2) integrated them in the GitHub source as submodules. However, the user may also obtain their source files directly from the projects' repositories. Both tools can be built using standard C++ toolchains[13] via their respective `Makefile` projects. For convenience, we provide scripts `lib/build_z3.sh` and `lib/build_einsim.sh` that the user may refer to for building the dependencies in-place.

The primary C++ application, called `harp-artifacts`, is built using the top-level `Makefile` project. After building Z3 within the subdirectory `lib/z3` (which installs Z3's C++ headers and library within that directory), the top-level `Makefile` will build `harp-artifacts` within the top-level directory.

[13] Note that C++17 is required to build the latest version of Z3.

A.5 Experiment Workflow

The artifacts are used to run three different experiments: Fig. 2, Fig. 4, and Figs. 6-10. We have provided example scripts under `evaluations/` for automatically running the latter two; however, we recommend the reader to extend these scripts based on their own compute environment in order to parallelize the simulation tasks (discussed in §A.7).

A.5.1 Fig. 2: Motivational Data. This is a standalone Python script that runs in ≈10 seconds with 120 MB of memory using an Intel i7-7700HQ CPU @ 2.80GHz. The script replicates Fig. 2 in a PDF output file (or can generate an interactive Matplotlib figure per command-line arguments).

A.5.2 Fig. 4: Post-correction Probability. This experiment comprises two steps: (1) data generation from C++ Monte-Carlo simulations and (2) data analysis using Python scripts.

Step 1: Data generation. `harp-artifacts` simulates how different representative ECC functions (i.e., single-error correcting Hamming codes with randomly-generated parity-check matrices) affect ECC words that exhibit uniform-randomly generated pre-correction error patterns. Each simulated error pattern comprises a single Monte-Carlo simulation sample. The Python scripts then aggregate these samples as part of Step 2.

The `harp-artifacts` binary takes several command-line arguments that are used to configure the experiment:

- *Analysis*: The experiment to run, either `probabilities` for the Fig. 4 experiment or `evaluations` for the Figs. 6-10 experiment.
- *ECC dataword length (k)*: The length (in bits) of a single-error correcting Hamming code dataword. This parameter defines the type of ECC code that will be generated and simulated (i.e., its generator and parity-check matrix dimensions). Our evaluations are all shown with $k = 64$, though we verified that all results and conclusions are consistent with other representative values, such as $k = 128$ (§7.1.2).
- *Number of ECC codes*: The number of Hamming code instances to generate and simulate. Each code's generator and parity-check matrices are created uniform-randomly according to the random seed command-line parameter.
- *Number of ECC words*: The number of ECC words to simulate for each randomly-generated ECC code.
- *Random seed*: The random seed to use for the first ECC code generated. The random seed is then incremented when generating each subsequent ECC code.

To run the simulations for this experiment, the `analysis` parameter should be provided as `probabilities`. §A.7 summarizes different configurations for the remaining parameters and their expected runtime and memory usage impact. All simulation output will be

given on `stdout`, which must be redirected to a text file to pass the data to the Python analysis scripts in the next step.

Step 2: Data analysis. The Python script `script/figure_4-parse_postcorrection_probabilities_data.py` accepts an input file containing the raw output from the previous step. The script will then parse, aggregate, analyze, and plot the data using `matplotlib` (either interactively or saved to a PDF file, based on a command-line switch).

A.5.3 Figs. 6-10: Profiler Evaluation.

This experiment runs in two parts: (1) data generation from C++ Monte-Carlo simulations and (2) data analysis using Python scripts. The experiment workflow is nearly identical to that of §A.5.2.

Step 1: Data generation. All `harp-artifacts` command-line parameters operate the same way as in §A.5.2, except the `analysis` parameter must be given as `evaluations` for this experiment.

In `evaluations` mode, `harp-artifacts` simulates the coverage achieved by the different profiling mechanisms that we evaluate in §7. These simulations are extremely time consuming due to the complexity of calculating error coverage, which requires a large number of computations, including repeated SAT solver invocations. §A.7 provides the expected runtimes for different configurations that the reader may use to estimate a viable configuration based on their available compute resources.

Step 2: Data analysis. The Python script `script/figures_6to10-parse_evaluation_data.py` accepts an input file containing the raw output from the previous step and is run the same way as the script in §A.5.2. Each figure is output either interactively or in individual PDF files based on a command line switch.

A.6 Evaluation and Expected Results

To replicate the results in this paper, it suffices to run each of the experiments as discussed in §A.5. Note that, as we discuss in §A.7, our full evaluation configuration is long-running due to the large number of Monte-Carlo samples that we simulate. It is *not* necessary to simulate this many samples to replicate our results; even with relatively few samples, the data yields similar conclusions. In general, we leave the reader to determine how many samples they can realistically simulate based on their available compute resources. To facilitate this, §A.7 provides runtime estimates measured during our own evaluations.

A.7 Experiment Customization

As §A.5.2 describes, `harp-artifacts` has independent command-line parameters to control the number of ECC codes and ECC words simulated. Using the ECC code parameter, it is possible to parallelize the simulations across different invocations of `harp-artifacts` (e.g., as independent jobs on different compute nodes). For example, simulating 100 ECC words for each of 1000 ECC codes can be done by simulating 10 ECC codes per job across 100 independent jobs (and incrementing the random seed by 10 for each subsequent job to avoid repeating the same experiment). The Python data analysis scripts will aggregate the raw data, regardless of how the ECC codes are partitioned.

A.8 Estimating Runtime and Memory Usage

Running the experiments for Fig. 4, and Figs. 6-10 requires accounting for available compute and memory resources. In this section, we discuss different configurations of `harp-artifacts` and how they impact runtime and memory usage when running on a compute cluster comprising Intel Xeon Gold 5118 systems.

A.8.1 Fig. 4 Estimation.

Table 3 summarizes the expected runtime and disk usage of a *single task* for different `harp-artifacts` configurations when running the Fig. 4 experiment. In the evaluations we present in §7, we run 16 instances of the "Evaluated" configuration shown in Table 3, amounting to a total of 74 GB of disk usage and approximately 1 CPU-day of total execution time. In general, the Fig. 4 simulations dump a large amount of raw data. However, the runtime and memory usage are relatively low, so the reader may feasibly replicate our full evaluation configuration.

Configuration	K	ECC codes	ECC words	Runtime	Disk Usage
Reduced	64	10	100	≈30 sec.	4.8 MB
Evaluated	64	100	10000	≈90 min.	4.6 GB

Table 3: Estimated runtime and disk usage for one instance of `harp-artifacts` when running the Fig. 4 experiment.

A.8.2 Figs. 6-10 Estimation.

Table 4 summarizes the expected runtime and disk usage of a *single task* for different `harp-artifacts` configurations when running the Figs. 6-10 experiments. In the evaluations we present in §7, we run 256 instances of the "Evaluated" configuration shown in Table 4, amounting to a total of 3.4 GB of disk usage and approximately 14 CPU-years of total execution time. In general, the memory and disk usage for `harp-artifacts` is negligible in this experiment, but the runtime can be a considerable limitation.

Configuration	K	ECC codes	ECC words	Runtime	Disk Usage
Reduced	8	1	1	≈10 sec.	-
Reduced	16	1	1	≈30 sec.	-
Reduced	32	1	1	≈5 min.	-
Reduced	64	1	1	≈30 min.	-
Evaluated	64	10	100	≈20 days	13 MB

Table 4: Estimated runtime and disk usage for one instance of `harp-artifacts` when running the Figs. 6-10 experiment.

As we mention in §A.6, it is not necessary to run the full "Evaluated" configuration to replicate our results. By the nature of Monte-Carlo simulation, simulating more ECC codes and ECC words improves the accuracy of the final estimates. However, the end conclusions are already apparent with relatively few samples (albeit with more noise in the data). Therefore, if running the full "Evaluated" configuration is infeasible, we recommend running a reduced configuration based on the available compute resources. Note that the average runtime scales linearly with the number of samples because the same computation is performed for each sample. For example, reducing the "Evaluated" configuration to 1 ECC code per task would reduce the average runtime by a factor of 10, resulting in jobs that comple in approximately 48 hours.

We find that the Python analysis for the full evaluated configuration (i.e., all 3.4 GB of output) takes approximately 3 hours and consumes 40 GB of memory. We observe that the memory usage is roughly linear in the number of evaluated ECC codes and words.