Implicit Memory Tagging: No-Overhead Memory Safety Using Alias-Free Tagged ECC

Michael B. Sullivan
NVIDIA
Austin, TX, USA
misullivan@nvidia.com

Mohamed Tarek Ibn Ziad
NVIDIA
Westford, MA, USA
mtarek@nvidia.com

Aamer Jaleel
NVIDIA
Westford, MA, USA
ajaleel@nvidia.com

Stephen W. Keckler
NVIDIA
Austin, TX, USA
skeckler@nvidia.com

ABSTRACT

Memory safety is a major security concern for unsafe programming languages, including C/C++ and CUDA/OpenACC. Hardware-accelerated memory tagging is an effective mechanism for detecting memory safety violations; however, its adoption is challenged by significant meta-data storage and memory traffic overheads. This paper proposes Implicit Memory Tagging (IMT), a novel approach that provides no-overhead hardware-accelerated memory tagging by leveraging the system error correcting code (ECC) to check for the equivalence of a memory tag in addition to its regular duties of detecting and correcting data errors. Implicit Memory Tagging relies on a new class of ECC codes called Alias-Free Tagged ECC (AFT-ECC) that can unambiguously identify tag mismatches in the absence of data errors, while maintaining the efficacy of ECC when data errors are present. When applied to GPUs, IMT addresses the increasing importance of GPU memory safety and the costs of adding meta-data to GPU memory. Ultimately, IMT detects memory safety violations without meta-data storage or memory access overheads. In practice, IMT can provide larger tag sizes than existing industry memory tagging implementations, enhancing security.

CCS CONCEPTS

• **Security and privacy → Hardware security implementation**; • **Hardware → Error detection and error correction**.

KEYWORDS

memory tagging, memory security, error correcting codes

ACM Reference Format:
Michael B. Sullivan, Mohamed Tarek Ibn Ziad, Aamer Jaleel, and Stephen W. Keckler. 2023. Implicit Memory Tagging: No-Overhead Memory Safety Using Alias-Free Tagged ECC. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589102

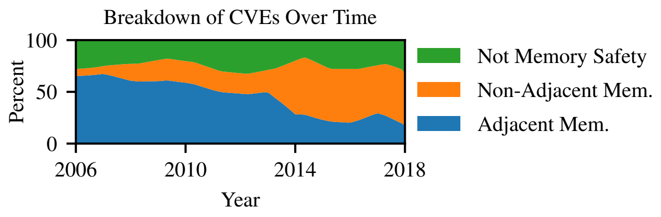


Figure 1: The prevalence of exploitable memory safety bugs in C/C++ code. Derived from slides 10 and 13 of [40].

1 INTRODUCTION

Business-critical servers, datacenters, high performance computing systems, and autonomous vehicles demand high dependability against both random hardware errors and security attacks. Memory safety is perhaps the largest security concern for unsafe programming languages, including C/C++ on CPUs [62, 67, 70] and CUDA/OpenACC for GPU programming [8, 39, 49]. Memory safety bugs persist in CPU programs despite the availability of various debugging tools, with Figure 1 showing that they have historically accounted for roughly 70% of the exploitable conditions in the Common Vulnerabilities and Exposures (CVE) database.

Memory tagging is an attractive hardware-accelerated mechanism for detecting memory safety vulnerabilities. It has been widely adopted in academic research [24] and in industry through SPARC Application Data Integrity (SPARC ADI) [48] and the ARM Memory Tagging Extension (ARM MTE) [4]. However, the adoption of memory tagging faces two primary challenges: (1) considerable meta-data storage and memory traffic overheads and (2) weak probabilistic security guarantees. By using a limited set of tags, memory tagging provides only probabilistic detection guarantees for non-adjacent buffer overflows, where out-of-bounds accesses occur beyond the immediate bounds of an allocation (*e.g.*, `a[d]`, with an attacker-controlled displacement `d`). Unfortunately, non-adjacent overflows are gaining more popularity in the recent years as shown in Figure 1. One way to strengthen the probabilistic memory tagging guarantees is to increase the tag size. While this approach provides heightened security, it comes at the expense of storage, memory traffic, or reliability overheads.

This paper proposes Implicit Memory Tagging (IMT), a scheme to provide hardware-accelerated memory tagging with no storage or memory traffic overheads. IMT leverages the system error correcting code (ECC) to check for the equivalence of a memory tag in addition to its regular duties of detecting and correcting data errors. IMT relies on a novel class of ECC codes we call Alias-Free Tagged ECC (AFT-ECC) that can unambiguously identify tag mismatches in the absence of data errors, while maintaining the efficacy of ECC when data errors are present. In contrast with prior work that either uses very small tags or offers probabilistic equivalence checking [14, 28, 59], IMT can perfectly diagnose tag mismatches with a maximum-length tag. The error detection capabilities of ECC are maintained by AFT-ECC against severe data errors, and single-bit data errors can be corrected.

Implicit Memory Tagging can be applied to any device with an ECC-protected memory hierarchy. We evaluate IMT on GPUs due to the growing importance of GPU memory safety and the cost associated with adding meta-data to GPU memory. As CPUs progressively employ hardware-based memory safety mitigations and GPUs increasingly can touch more application memory [16, 55, 56], GPU-side exploits become a valuable target [8, 39, 49]. However, GPU memory capacity is relatively small, it cannot be increased easily [7], and adding meta-data consumes relatively-scarce memory bandwidth. Accordingly, we use IMT to tag GPU memory with no extra meta-data by leveraging the existing GPU ECC.

The main contributions of this paper are as follows.

- We propose a novel and general class of ECC codes we call Alias-Free Tagged ECC (AFT-ECC). These codes embed a maximum-length tag in the ECC check-bits, and they are able to unambiguously identify tag mismatches while preserving single-bit data error correction. We demonstrate how to generate a parity-check matrix with alias-free tag bits and the limits of how large a tag can be with the alias-free property. For most common error-correcting codeword sizes, the maximum tag size is one fewer bit than the ECC redundancy.
- We propose Implicit Memory Tagging (IMT) to provide hardware-accelerated memory safety with no storage or memory traffic overheads. Implicit Memory Tagging uses AFT-ECC to check for the equivalence of a memory tag in addition to its regular duties of detecting and correcting data errors.
- We demonstrate that IMT outperforms alternative hardware-accelerated memory tagging approaches in performance, security, and reliability. Our approach incurs no performance penalties on top of those already imposed by ECC protection, it does not degrade the error correction or detection capabilities of the underlying ECC code, and in practice it can provide a larger tag size than existing industry memory tagging implementations.

2 BACKGROUND

2.1 Basic Terminology

This paper aims to achieve high reliability, availability, and both spatial and temporal memory safety. *Reliability* involves preventing silent data corruption (SDC) through error detection, while *availability* entails error correction to ensure rare interruptions and high machine uptime. Spatial memory safety guarantees that memory allocations are accessed within their intended bounds, while temporal memory safety ensures that allocations are only used during their lifetime. Undetected buffer over-/under-flows can compromise spatial memory safety, whereas use-after-frees (dangling pointers) can violate temporal memory safety.

2.2 Threat Model

We assume a comparable threat model to prior work on memory safety for GPUs [9, 11, 33] and memory tagging on CPUs [4, 48], where the victim program has one or more memory safety vulnerabilities and that the attacker's goal is to use malicious program inputs to obtain arbitrary read or write access to memory. The attacker can then use these read and write capabilities for various malevolent ends, such as information leakage, data corruption, and/or remote code execution. Additionally, we assume that the attacker is aware of the memory tagging mechanism but they cannot modify the program's binary image. Finally, we assume that the GPU driver and hardware are trusted and tamper-resistant. Hence, we consider attacks that exploit hardware vulnerabilities, such as RowHammer [29] and side-channel attacks [73], to be outside the scope of this work.

2.3 Memory Tagging

Memory tagging is a technique to detect memory safety violations in low-level programming languages, such as C/C++ and CUDA/OpenACC. It works by assigning a fixed-size *lock tag* to each memory granule and storing an identical *key tag* in the upper bits of data pointers. During execution, a memory safety violation is reported upon a tag mismatch (TMM) between the lock tag in memory and the key tag of the pointer used to access it. Tags can be reused to protect an unrestricted number of allocations. This is especially important for massively-parallel GPU programs that use dynamic per-thread memory allocations [46]. The *tag size*, TS, and *granule size*, TG, impact the security, reliability, and performance of memory tagging. Existing implementations tend to use small tag sizes that provide weak security guarantees. For instance, SPARC ADI [48] assigns a TS = 4-bit tag to each TG = 64B memory granule, while ARM MTE [4, 5] uses a TS = 4-bit tag at the TG = 16B granularity.

Efficient memory tagging schemes such as SPARC ADI or ARM MTE are mostly implemented in hardware, with key and lock tag equivalence checking for every memory access. However, there is also a software component that tags memory objects when they are allocated and clears the tags when they are freed using specialized ISA instructions. Our work optimizes the storage, movement, and checking of the lock tag, which is responsible for most of the overheads associated with memory tagging. We do not provide detailed information about the software runtime or ISA changes because these components require no modification for IMT.

2.4 GPU Memory Hierarchy

This paper applies IMT to GPUs using NVIDIA terminology. Figure 2 shows the memory hierarchy of a discrete GPU. GPUs rely on many high-bandwidth memory channels to support concurrent DRAM accesses from a massive number of threads. Each DRAM channel is connected to one or more L2 slices; these L2 slices are

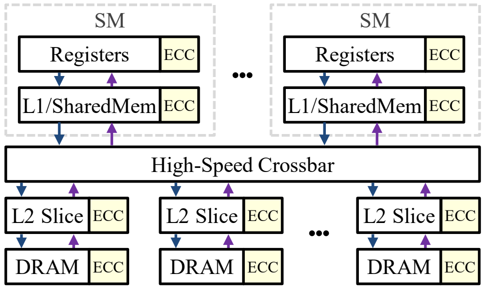


Figure 2: The memory hierarchy of NVIDIA GPUs.

shared among all streaming multiprocessors (SMs) through a large high-speed crossbar. Each SM has its own L1 cache, which can also be configured to operate as a shared memory scratchpad. The minimum access granularity of GPU memory (GDDR [19, 20, 22] or HBM [21, 23]) is 32B. Thus, we assume that GPUs support a fine-grained 32B access granularity throughout the memory hierarchy using sectored caches; this is corroborated by prior work on NVIDIA GPUs [25] and work that shows fine-grained GPU accesses to be performant for important workloads [53]. All major storage structures are ECC-protected in compute-class GPUs, and we assume that a single codeword is formed for each 32B sector of data, to align with the memory access granularity.

2.5 Error Detecting and Correcting Codes

An error correcting code (ECC) detects and possibly corrects errors using redundant values generated algorithmically from the data. An (N, K) code has N total bits with K data-bits and $R = (N-K)$ check-bits. A valid ECC data and check-bit pair is called a *codeword*. *Encoding* is the process of generating a codeword from data, while *decoding* is the process of detecting and (possibly) correcting errors in a codeword to restore the original data. Encoding and decoding are typically performed in low-latency fixed-function hardware.

An ECC code with R bits of redundancy can provide single-bit error correction on up to $N = 2^R - 1$ bits (including the check-bits, themselves). The maximal data size is therefore a non-power-of-2 bits, and it is common to *shorten* an ECC code by having fewer than the allowable data bits. The SDC risk of shortened codes decreases with the number of data bits. A heavily shortened code (with more check-bits than is strictly necessary to provide single-bit error correction) can be used to decrease the SDC risk to low levels.

2.6 Tagged ECC

Alias-Free Tagged ECC builds upon prior work that proposes *Tagged ECC*, where a shortened ECC code is extended to also check for the equivalence of a tag. Four decades ago, Gumpertz was the first to describe the concept of tag checking with ECC [13, 14]. Sazeides *et al.* describe a similar scheme they call *error code tagging* [59]. Multiple works use a form of memory tagging to protect against random DRAM address errors [28, 58]. Alias-Free Tagged ECC improves upon this prior work by unambiguously detecting tag mismatches using a maximum-width tag. By maximizing the tag size, AFT-ECC provides the highest possible security guarantees for IMT.

2.7 ECC in GPUs

Compute-class GPUs use ECC to protect major storage structures including GPU DRAM, caches, and register files, as shown in Figure 2. The GPU memory hierarchy is purportedly protected using single-bit-error-correcting and double-bit-error-detecting (SEC-DED) ECC [1, 42], which requires at least 10 bits of redundancy to protect each 32B memory location. Redundancy is provisioned in DRAM at the byte granularity, however, due to structural constraints. Modern HBM3 memory provisions 2B of sideband rank-level ECC per 32B data access [23]. Embedded ECC with GDDR6 memory is implemented using the same 6.25% redundancy,[1] which we assume also offers 2B of ECC per 32B data access. This paper therefore evaluates two GPU ECC configurations: one with the minimum 10 bits of SEC-DED redundancy, and one with the DRAM-provided 16 bits of redundancy.

3 ALIAS-FREE TAGGED ECC

We propose a novel and general class of ECC codes called Alias-Free Tagged ECC (AFT-ECC) that perform tag equivalence checking using the ECC check-bits. Alias-Free Tagged ECC codes are designed to support large tags and unambiguously identify tag mismatches while preserving single-bit data error correction. We demonstrate how to generate a parity-check matrix with alias-free tag bits, the limits of how large a tag can be with the alias-free property, and recommend some concrete code designs below.

3.1 Linear Codes and the Parity Check Matrix

A linear block code generates its R check-bits using a linear combination of the data. An R×N *parity check matrix*, H, determines the allowed linear relations for a given code. Valid codewords are defined as those that fall in the null space of H, as shown by Equation 1a.[2] An N-bit codeword, c, could potentially suffer from an error, e, during storage, transmission, or decoding. In this case the codeword is corrupted to c_e ("codeword with error"), as shown in Equation 1b.

$$H * c^T = 0 \tag{1a}$$

$$c_e = c + e \tag{1b}$$

Upon receiving a possibly-erroneous codeword, the decoder first generates a *syndrome*, s, as shown in Equation 2. If the codeword is invalid, the syndrome will be non-zero. The magnitude of the syndrome depends only on the error. Error correction can take place for a restricted class of errors (*e.g.*, single-bit corruption) by constraining H such that each correctable error maps to a single unique syndrome. For example, to provide single-bit error correction, the constraints on the H matrix are simple: every column must be unique.

$$s = H * c_e^T = H * (\underbrace{c+e}_{\text{From Equation 1b}})^T = H * c^T + H * e^T = \underbrace{0}_{\text{From Equation 1a}} + H * e^T = H * e^T \tag{2}$$

A systematic code is one whose data and check-bit locations are fixed at design time; most practical ECC codes are systematic [12]. Structurally, a systematic code places further constraints on the

[1]We estimate the GDDR embedded ECC redundancy by inspecting an NVIDIA T4 GPU [43] using the `nvidia-smi` tool—with ECC enabled, the available device memory is 6.25% smaller than with ECC disabled.
[2]Arithmetic is being performed in a binary finite field (GF(2)), where single-element addition and multiplication correspond to XOR and AND operations, respectively.

parity check matrix. The canonical form of a systematic parity check matrix is shown in Equation 3. The check-bit columns are constrained to form the identity matrix. Thus, the systematic H matrix can be thought of as two *submatrices*: the R×K data submatrix (D_K) and the R×R identity submatrix (I_R).

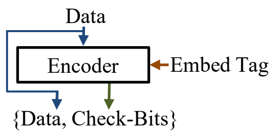
$$H_{\text{Systematic}} = \underbrace{\left(D_K \mid I_R \right)}_{N = K + R} \Big\} R \tag{3}$$

3.2 Tagged ECC

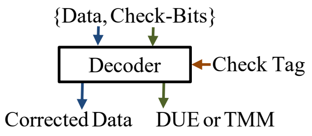
Tagged ECC is a construction where decoding also checks for the equivalence of a TS-bit tag [13, 14]. Tagged ECC introduces an additional R×TS parity check submatrix, T_{TS}, as shown in Equation 4. Such tagging is possible for any sufficiently-shortened ECC code.

$$H_{\text{Tagged}} = \underbrace{\left(T_{TS} \mid D_K \mid I_R \right)}_{N = TS + K + R} \Big\} R \tag{4}$$

Figure 3 shows the Tagged ECC encoder and decoder. Tagged ECC works by generating ECC check-bits using both the data and tag, but crucially it never stores the tag explicitly to memory. Rather, upon ECC decoding the equivalence of the encoded tag is checked against a reference tag that is given to the decoder.



(a) Tagged ECC Encoder.



(b) Tagged ECC Decoder.

Figure 3: Tagged ECC. The tag is encoded and decoded in ECC but it is never written to memory. DUE = Detectable-Uncorrectable Error; TMM = Tag Mismatch.

Tagged ECC Limitations: Previous Tagged ECC implementations are limited in two ways, each of which would reduce the security of memory tagging. Some implementations are restricted to small tags [14, 28, 59]. Others allow unrestricted tag size but provide only probabilistic tag mismatch guarantees (*e.g.* tag-hashed configurations from [13], or address parity [58]), meaning that some invalid tags remain undetected. Both approaches are unsuitable for tagged memory due to the security degradation caused by limited tag sizes or tag aliasing.

3.3 Alias-Free Tagged ECC Code Construction

Tagged ECC is an important concept which forms the theoretical basis of this work, but in its prior form it has limited applicability for memory tagging due to its limited tag size or probabilistic nature. This work proposes an improved code we call *Alias-Free Tagged ECC (AFT-ECC)* that maintains the following properties.

(1) **Alias-Free:** A multi-bit error can cause an erroneous codeword to *alias* to a valid-yet-wrong codeword, evading detection. A tag mismatch with TS > 1 manifests to the decoder as a multi-bit error, but we design the tag submatrix to be alias-free, ensuring complete tag mismatch detection in the absence of a data error.
(2) **Single-Bit Error Correction:** AFT-ECC maintains single-bit error correction against data errors.
(3) **Maximal Tag Size:** AFT-ECC supports the largest possible tag size with the above properties.

We denote the set of all linear combinations of the columns of the tag submatrix T_{TS} by $\mathbb{T}$. Alternatively, this can also be thought of as the column space of T_{TS}.

Ensuring Alias-Free Tags: A tag submatrix is *alias-free* if no tag mismatch remains uncaught (*i.e.* if $0 \notin \mathbb{T}$). In other words, all columns of an alias-free tag submatrix must be linearly independent. Thus, any left-invertible tag submatrix with full column rank has the alias-free property.

Maintaining Single-Bit Error Correction: To maintain single-bit error correction, the column space of T_{TS} must be not intersect with any correctable error syndrome. Thus, to maintain single-bit error correction, no column of the D_K or I_R submatrices should appear in the column space of T_{TS}.

Figure 4 visualizes the constraints placed on an AFT-ECC parity check matrix. To maintain single-bit error correction, the column space of T_{TS} ($\mathbb{T}$, yellow) must not intersect with the data and identity submatrices ($D_N \cup I_R$, light green). To ensure no tag aliasing, $\mathbb{T}$ must also not contain the all-0-syndrome ({0}, red). As we describe later, the syndrome of some multi-bit errors, including 2b errors (dark green), *do* overlap with $\mathbb{T}$, potentially leading to misattribution as a TMM unless they are properly diagnosed at higher system levels.

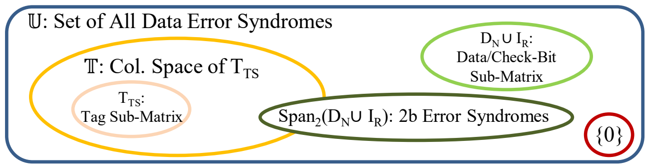

𝕌: Set of All Data Error Syndromes
𝕋: Col. Space of T_TS
T_TS: Tag Sub-Matrix
D_N ∪ I_R: Data/Check-Bit Sub-Matrix
Span_2(D_N ∪ I_R): 2b Error Syndromes
{0}

Figure 4: A set-intersection visualization of the Alias-Free Tagged ECC syndrome spaces.

3.4 Alias-Free Tag Size Limits

It follows from the alias-free property that the maximum tag submatrix width is $TS \leq R$. If $TS > R$, the R×TS tag submatrix will be non-square with more columns than rows. Some linear dependence must exist along the larger dimension of a non-square matrix, and thus with $TS > R$ the alias-free property cannot hold.

Because an alias-free submatrix maps each tag error to a unique syndrome, the dimension of the column space of T_{TS} is $\dim(\mathbb{T}) = 2^{TS} - 1$. Thus, error correction cannot be maintained when $TS = R$, as the column space of T covers all possible non-zero syndromes, making it impossible to have unique correctable bit-error syndromes. With $TS < R$, however, there is the possibility of maintaining single-bit error correction up to a certain data size. To maintain single-bit error correction, $\dim(\mathbb{T})$ must be small enough to leave at least K+R of the $2^R - 1$ possible syndromes free. An upper bound on the alias-free tag size with single-bit error correction is thus given by

Equation 5a. Solving for the maximum tag size yields Equation 5b.

$$2^R - 1 - (2^{TS} - 1) \geq K + R \tag{5a}$$

$$TS \leq \lfloor \log_2(2^R - K - R) \rfloor \tag{5b}$$

Figure 5 evaluates the bound given by Equation 5b at various data sizes and ECC redundancies. Each point is colored according to the maximum value of TS at that (K, R). Unshortened codes are marked with a left-facing triangle, and white points cannot support error correction. The behavior is intuitive—an unshortened code cannot support any tag; with one bit of shortening, at most TS = 1-bit is possible (dark blue). As the degree of shortening increases, so does the maximum TS. For common SEC-DED codewords with a power-of-2 data size, the maximum tag size is one fewer bit than the ECC redundancy. We have marked the two codeword sizes we evaluate for Implicit Memory Tagging with stars. At R = 10, IMT supports at most a TS = 9-bit tag; at R = 16, the maximum TS = 15 bits.

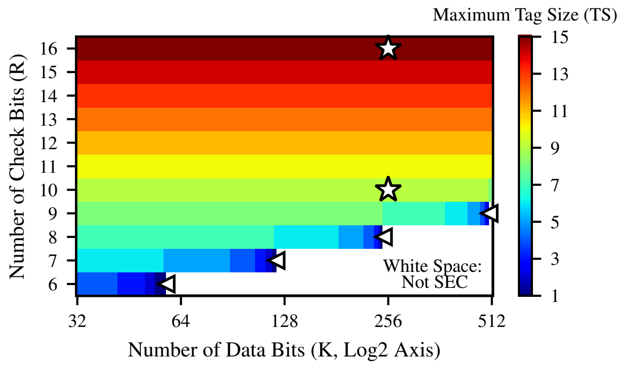


Figure 5: The maximum Alias-Free Tagged ECC tag size (TS) across various data sizes and ECC redundancies.

3.5 Parity-Check Matrix Recommendations

An attractive strategy for selecting the tag and data submatrices is to use a left-invertible matrix with all-even-weight columns for the tag submatrix and unique all-odd-weight columns for the data submatrix. This ensures that the tag submatrix is alias-free, and it maintains double-bit detection and single-bit correction of data errors.

Double-Bit Data Error Detection: Parity check matrices with unique all-odd-weight columns (such as Hsiao codes [15]) are SEC-DED, because the XOR of any two odd-weight columns gives an even-weight syndrome, and no aliasing is possible between an even-weight syndrome and exclusively odd-weight columns.

Maintaining Single-Bit Data Error Correction: The column space of any all-even-weight tag submatrix is all-even. No aliasing is therefore possible between the even-weight T_{TS} column space and the odd-weight D_K or I_R submatrices. Thus, single-bit error correction is maintained.

Minimizing Hardware Overheads: One further recommendation is to use all-weight-2 columns for the tag submatrix, to minimize the additional decoder area and delay.

Selected Tag Submatrices: Given the selection recommendations above, one can analytically find attractive tag submatrices without an extensive search. Equation 6 gives a T_{TS} recommendation for AFT-ECC with R = 16 bits of ECC redundancy, which can support up to a TS = 15 bit tag for most data sizes. The number of 1s is minimized in this alias-free tag submatrix (reducing decoder area), as well as the maximum number of 1s per row (reducing delay).

$$T_{TS}\ (R \leq 16,\ TS \leq 15) = \begin{pmatrix} 000000000000001 \\ 000000000000011 \\ 000000000000110 \\ 000000000001100 \\ 000000000011000 \\ 000000000110000 \\ 000000001100000 \\ 000000011000000 \\ 000000110000000 \\ 000001100000000 \\ 000011000000000 \\ 000110000000000 \\ 001100000000000 \\ 011000000000000 \\ 110000000000000 \\ 100000000000000 \end{pmatrix} \tag{6}$$

$$\underbrace{\hphantom{000000000}}_{(R=10,\,TS=9)}$$

Shortening the Tag Submatrices: The full tag submatrix in Equation 6 supports the largest single-error-correcting tag size (TS = R − 1 = 15), which is valid only up to a certain data size. If more data bits are required, one can lessen the TS of a new tag submatrix by removing columns. As any subset of columns from an alias-free tag submatrix is also alias-free, it does not matter which columns are removed. The columns of T_{TS} shown in Equation 6 are also sorted to allow submatrix formation for lower ECC redundancies by retaining only the R lowest rows. An example with (R = 10, TS = 9) is highlighted in blue—the first 10 rows and 9 columns can be used to create a valid tag submatrix at this size.

Selected Data Submatrices: The nature of our AFT-ECC tag submatrix construction makes it easy to find an AFT-ECC data submatrix—any all-odd-column D_K is sufficient. That being said, a search of the code space can provide modest benefits in some metrics. In this paper we use minimum odd-weight-column D_K submatrices that are selected via a genetic algorithm to minimize the maximum number of 1s per row and to maximize 3-bit error detection.

3.6 AFT-ECC Behavior and Constraints

Alias-Free Tagged ECC is a general mechanism that may have many uses—this paper focuses on the important problem of memory tagging for security; possible other uses are described later in Section 7. Alias-Free Tagged ECC provides tag equivalence checking while maintaining the correction and detection capabilities of ECC, so long as the use-case follows two constraints.

(1) **Fatal TMM:** To maintain the error detection capabilities of ECC, a tag mismatch (TMM) must be fatal, or it must include some recovery action that also works for recovering from data errors (*e.g.*, rollback and restart from an error-free checkpoint). This is because, in rare multi-bit error cases, a severe data error could be misidentified as a tag mismatch.

(2) **No Tag Values:** Alias-Free Tagged ECC provides implicit tag equivalence checking, but it does *not* explicitly store the tag values. Thus, it is impossible to extract the tag value assigned to an AFT-ECC codeword using without corrupting the extracted tag in the presence of some multi-bit data errors.

If the fatal tag mismatch constraint holds, AFT-ECC has the following behavior. In the absence of a data error, all tag mismatches are detected and properly attributed (TMM = 100%). In the absence of a tag mismatch, single-bit data errors are corrected. Because multi-bit data errors are detected with the same probability as untagged ECC, there is no increased risk of silent data corruption. However, there is some *misattribution* risk of AFT-ECC reporting a multi-bit DUE as a TMM. We quantify this misattribution risk later in Section 5.3 and describe an optional driver-side diagnosis procedure to avoid any misattribution in Section 4.3.

Importantly, with AFT-ECC there is no risk of reporting a TMM as a DUE, which is desirable for IMT—due to the rarity of data errors and the fact that attackers can induce tag mismatches, the expected rate of TMMs far exceeds that of DUEs. If attacker-induced TMMs could be misattributed as DUEs, it would render the reported DUE rate essentially useless for reliability field studies. Also, if a TMM could be misattributed as a DUE, an attacker could maliciously trigger the GPU persistent error retirement mechanisms [44, 45] to make them unusable.

There is a vanishingly small possibility of *both* a tag mismatch and a random error affecting the same codeword.[3] This rare event is considered by some [28] but not all [13, 14, 51, 59] related evaluations. In the simultaneous presence of both events, the effective number of erroneous bits is the sum of the tag mismatch and data errors. In this case, AFT-ECC maintains high probabilistic error detection, but it cannot guarantee detection of all 1 or 2-bit data errors when combined with an arbitrary tag mismatch.

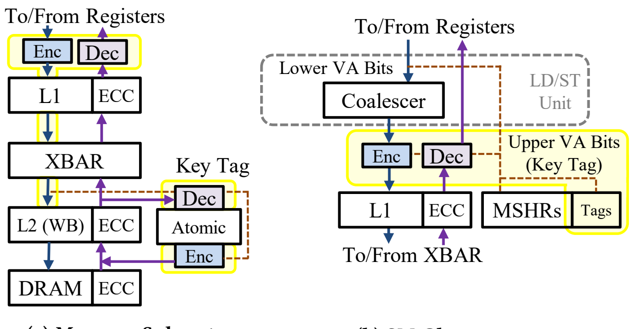
4 IMPLICIT MEMORY TAGGING

We propose Implicit Memory Tagging (IMT), a scheme to provide hardware-accelerated memory tagging on GPUs. IMT uses AFT-ECC to overcome the limitations of prior tagged memory approaches and provide memory safety with no storage or performance overheads and no additional risk of silent data corruption.

4.1 Prior Approaches and Limitations

ECC Stealing: At its core, memory tagging associates a TS-bit tag with each TG-bit granule of memory. This is similar to how each ECC codeword associates R check-bits with K bits of data. In fact, ECC redundancy could possibly be "stolen" to instead provide tagging for memory safety. While the details are proprietary and unknown, this may be the way that SPARC ADI tags memory—by placing its 4 tag bits in redundant sideband storage originally envisioned for ECC check-bits [47, 60]. Similarly, IBM POWER9 systems may place a 1-bit tag in stolen ECC storage [32]. Placing the lock tags in sideband ECC storage has no performance overheads, but it comes at the cost of reduced reliability. As we show in Section 5.3, each bit stolen from ECC roughly doubles the silent data corruption risk. Some prior tagged memory implementations use widened machine words (or reduced data dynamic ranges) to store tags [24, 31, 76]. We consider this as largely equivalent to ECC stealing, because this additional sideband storage *could* be used for ECC.

[3]One exception to this might be if an attacker could induce RowHammer errors while crafting a tag mismatch. As mentioned in Section 2.2, RowHammer attacks are out-of-scope for this paper and should be dealt with by other means.



(a) Memory Subsystem **(b) SM Changes**

Figure 6: IMT propagates tag information along with ECC check-bits. Only ECC encoder/decoder changes, widened address busses, and some supplemental L1 MSHR meta-data are needed (changes are highlighted).

Tag Carve-Out: As an alternative to ECC stealing, memory tagging can rely on a dedicated, densely-packed carve-out of tag bits [24]. To maintain high performance without sideband meta-data such as ECC, the tag bits should be cached on chip, either in a dedicated cache hierarchy (which adds area and complexity), or as non-data information in the regular cache hierarchy (risking cache pollution). It is likely that ARM MTE uses a tag carve-out to implement memory tagging [4]. Using a tag carve-out does not have any reliability implications, but it requires additional storage capacity (both in off-chip and on-chip memory) and redundant memory traffic to store, fetch, and cache the tags.

4.2 Implicit Memory Tagging Design

Rather than stealing ECC bits to explicitly store each lock tag, IMT uses AFT-ECC to check for the equivalence of the key and lock tags in addition to detecting and correcting data errors. This strategy preserves the performance and storage advantages of ECC stealing, without its reliability degradation.

End-to-End ECC and Widened Address Busses: Implicit Memory Tagging embeds the lock tag in the ECC check-bits and the key tag in the upper address bits. In practice, this requires that the on-chip address busses down to the L2 cache be widened to hold the key tags associated with a memory operation, unless the address busses are sufficiently over-provisioned already. The key tag is extracted from the address and passed to all ECC encoding and decoding points. End-to-end ECC (as shown in Figure 6a) is one possible implementation, where ECC check-bits are propagated along with the data, and decoding is only performed shortly before use. Alternatively, the key tag could be passed to all ECC encoders and decoders in the path to main memory. End-to-end ECC must be used past the point of the first write-back cache, however, as upon a dirty writeback the ECC-embedded tag value cannot be safely extracted from the AFT-ECC check-bits.

Support for Atomic Memory Operations: GPUs support near-memory atomic operations that are serviced in the L2 cache. To guard against L2 storage errors, the atomic datapath presumably is surrounded by an ECC decoder and encoder. Thus, IMT must pass the key tag to these encoders and decoders through widened address busses, as seen in Figure 6a.

SM Changes: Several small augmentations are required in the SM, as seen in Figure 6b. The upper virtual address (VA) bits are extracted from the per-thread address stream before coalescing—this ensures that the memory coalescing unit does not split apart neighboring memory addresses with differing key tags. The key tags are stored in the widened address bits of whatever structure the L1 uses for miss-status handling. When data comes back from a lower level of memory, the L1 MSHR structure provides the tag to the AFT-ECC decoder.

4.3 Software Interface and Precise Diagnosis

Figure 7 shows the IMT hardware-software interface. When a fatal error occurs, the faulting address, key tag, and ECC syndrome are sent to the GPU driver for further diagnosis. This is similar to existing CPU error-reporting systems [2], and this information may already be passed to the GPU driver. Driver-side error diagnosis is performed in software to extract a stored tag estimate and optionally a golden reference tag for the faulting memory location; these tags (along with the key tag) are used to differentiate between a TMM, DUE, or both.

Key Tag: Upon a fatal error (TMM or DUE), the key tag is taken from the upper address bits and sent to the driver.

Lock Tag: IMT can extract a lock tag estimate upon a TMM—each tag error maps to a unique syndrome, which if XORed with the key tag will give the lock tag. This can be performed by XORing the key tag with the error pattern given by a $2^R - 1$ entry syndrome lookup table. If the syndrome of a fatal error is not in the tag syndrome table, the error is a DUE and the tag estimate is set to an always-invalid value. If the fatal error is in the tag syndrome table, the error is either a TMM (in which case the tag estimate is correct) or a misattributed DUE (in which case the lock tag estimate is corrupted). Optionally, DUE misattribution can be avoided by performing precise diagnosis in the GPU driver, as described below.

(Optional) Reference Tag: The GPU driver has knowledge of all global memory allocations, and it can be optionally augmented to precisely track the tags associated with each memory object (perhaps through a storage-efficient tree structure [26, 50]). Upon a fatal error, the driver traverses this structure to lookup a reference tag for the faulting memory location. This relatively-expensive tag lookup only needs to be performed in the rare case of a fatal error, so it is not performance critical.

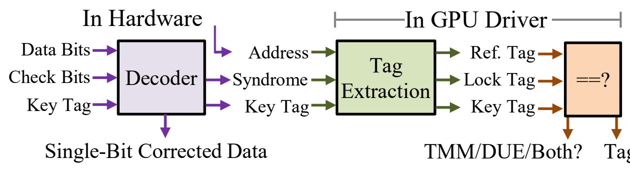


Figure 7: Error and tag-mismatch analysis and reporting in IMT. Correctable data errors are handled completely in fixed-function logic. Fatal errors (TMMs or DUEs) are diagnosed and logged in software. Key Tag = Tag in Upper Pointer Bits; Lock Tag = Tag Extracted from Syndrome Lookup; Ref. Tag = Driver-Side Reference Tag for the Faulting Address.

(Optional) Precise Diagnosis with Reference: Equation 7 shows the logic to precisely differentiate a TMM from a DUE using the reference tag. In no case will the key and extracted lock tags be equal—if they were, no fatal error would have been flagged by the ECC decoder. Upon a TMM (no data error), the true stored lock tag will be extracted and it will match the driver's reference tag. Upon a DUE (no tag mismatch), the lock tag will be corrupted but the key tag and reference tag should be equal. If none of the three tags match, both a multi-bit data error and a tag mismatch must have occurred simultaneously.

$$\text{Diagnosis} = \begin{cases} \text{TMM}: & \text{Ref} \neq \text{Key} \quad \text{and} \quad \text{Ref} = \text{Lock} \\ \text{DUE}: & \text{Ref} = \text{Key} \quad \text{and} \quad \text{Ref} \neq \text{Lock} \\ \text{BOTH}: & \text{Ref} \neq \text{Key} \quad \text{and} \quad \text{Ref} \neq \text{Lock} \end{cases} \tag{7}$$

Debug Mode: The fatal tag-mismatch constraint mentioned in Section 3.6 arises from the need to maintain immediate error detection of severe multi-bit errors. Design-time testing and debugging may benefit from passively logging TMMs, however, and we envision a privileged mode (*e.g.* through `nvidia-smi`) where the GPU is reconfigured to asynchronously raise DUEs and to consider tag mismatches non-fatal. This violates the synchronous error containment guarantees of modern GPUs [44], but it allows tag mismatches to be logged without destroying the parent CUDA context.

4.4 IMT Configurations for GPU

Implicit Memory Tagging uses the unused upper bits of a data pointer to store the key tag. GPUs are designed to support a variety of CPUs with different virtual address ranges. The most common virtual address spaces for GPUs are a 48b VA with `x86_64` CPUs, a 49b VA with ARM CPUs, and a 46b VA with IBM POWER9 CPUs. In this paper, we assume a 49b virtual address space, which has room for up to 15-bit key tags.[4] We propose two IMT configurations, with differing ECC redundancies. IMT-16 utilizes the full GPU-provided 16 bits of redundancy per 32B access for ECC. It embeds a 15-bit lock tag at the 32B codeword granularity. As Implicit Memory Tagging is compatible with any sufficiently-shortened binary ECC, we also introduce IMT-10, which features the minimum 10 bits of SEC-DED redundancy. IMT-10 embeds a 9-bit tag per codeword.

5 EVALUATION

The following evaluation shows that Implicit Memory Tagging outperforms alternative hardware-accelerated memory tagging approaches in performance, reliability, and security. Furthermore, we show that the additional encoding and decoding hardware overheads from Alias-Free Tagged ECC are nominal.

Table 1 summarizes our findings, comparing and contrasting Implicit Memory Tagging with prior memory tagging approaches. The far left columns show ECC stealing and tag carve-out configurations that are similar to SPARC ADI [48] and ARM MTE [4], with a limited TS = 4-bit memory tag. We show two Implicit Memory Tagging variants—IMT-10 and IMT-16—with 10 and 16 bits of redundancy, respectively, alongside ECC stealing and tag carve-out

[4]Very recent `x86_64` systems can operate with a 57-bit VA [3, 18], leaving only 7 unused upper bits. IMT could embed a 7-bit key tag on such systems, but we defer this evaluation since most GPUs lack 57-bit VA support.

Table 1: A comparison of alternative memory tagging implementations.

	ECC Stealing (SPARC ADI)	Tag Carve-Out (ARM MTE)	ECC Stealing Iso-Security-10	Tag Carve-Out Iso-Security-10	Implicit Memory Tagging (IMT-10)	ECC Stealing Iso-Security-16	Tag Carve-Out Iso-Security-16	Implicit Memory Tagging (IMT-16)
Tag Granularity (TG)	32B*	16B	32B	32B	32B	32B	32B	32B
Tag Size (TS)	4b	4b	9b	8b†	9b	15b	16b†	15b
Tag Store Overhead	0%	3.125%	0%	3.125%	0%	0%	6.250%	0%
Avg. Perf Overhead‡	None	1–4%	None	1–4%	None	None	2–7%	None
Max Perf Overhead‡	None	32%	None	32%	None	None	43%	None
ECC Redundancy	12b	16b	1b	10b	10b	1b	16b	16b
Error Correction	Yes	Yes	No	Yes	Yes	No	Yes	Yes
Added SDC Risk§	15.76×	None	1.917×	None	None	120.0×	None	None
Num. Tags (glibc¶)	14	14	510	254	510	32766	65534	32766
Adj. Security (glibc¶)	92.857%	92.857%	99.804%	99.607%	99.804%	99.997%	99.998%	99.997%
Non-Adj. Sec. (glibc¶)	92.857%	92.857%	99.804%	99.607%	99.804%	99.997%	99.998%	99.997%
Num. Tags (Scudo¶)	7	7	255	127	255	16383	32767	16383
Adj. Security (Scudo¶)	100%	100%	100%	100%	100%	100%	100%	100%
Non-Adj. Sec. (Scudo¶)	85.714%	85.714%	99.608%	99.212%	99.608%	99.994%	99.997%	99.994%

* SPARC ADI protects at a TG=64B granularity, but we adjust this to match the 32B codeword size of GPU ECC.
† Our Iso-Security tag carve-out configurations are byte aligned to avoid expensive unaligned accesses.
‡ Workload runtime overheads; estimated using simulation, see Section 5.2 (Figure 8b).
§ Severe multi-bit SDC risk relative to 10b (IMT-10) or 16b (IMT-16) SEC-DED ECC; estimated using software decoders, see Section 5.3 (Figure 9).
¶ Security guarantees depend on how the memory allocator retags memory. IMT is compatible with any allocator; two existing examples are shown.

configurations that are designed to offer roughly-equal security guarantees.[5] Red entries are particularly concerning. ECC stealing has no storage or performance overheads, but it comes at the cost of reliability—sacrificing error correction, having high SDC risk, or both. In comparison, a tag carve-out can support greater levels of security without reliability concerns. However, the tag carve-out does so with storage and performance overheads. By leveraging the ECC redundancy, Implicit Memory Tagging can scale to tag sizes that are costly for prior memory tagging approaches. Implicit Memory Tagging is able to preserve the performance benefits of ECC stealing without its reliability degradation, while also avoiding the storage and performance overheads of the tag carve-out.

Note that the ARM MTE configuration tags memory at a TG = 16B granularity whereas ECC stealing and IMT must match the 32B GPU ECC granularity. Using TG = 32B can cause memory footprint bloat for < 32B allocations, but GPU applications do not tend to have an abundance of small allocations and this does not appear to be of concern. Over the workloads we evaluate, significant bloat is seen only in programs with tiny memory footprints (for which high memory footprint bloat is acceptable). Workloads that use ≤ 1MB of total memory see a harmonic mean footprint bloat of 5.23%, with a maximum of 50%. Workloads that use > 1MB of memory see only a 0.21% average footprint bloat, with a maximum of 1.8%.

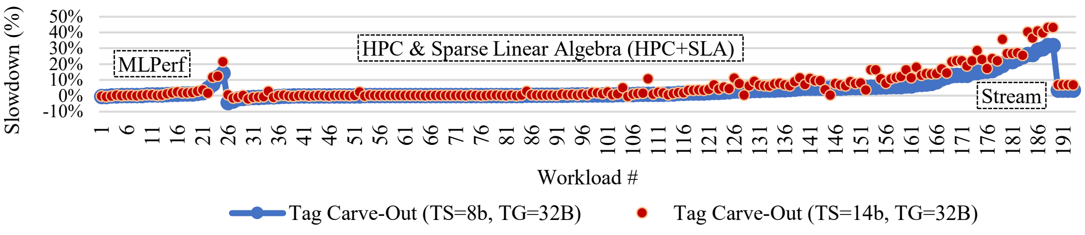
5.1 Methodology and Benchmarks

We evaluate the performance overheads of the tag carve-out on an NVIDIA GV100 GPU using the NVIDIA Architectural Simulator (NVAS) [71]. We carve out 1GB of the 32GB total physical memory for the ARM-MTE (TS = 4-bit, TG = 16B) and Iso-Security-10 (TS = 8-bit, TG = 32B) configurations, and 2GB physical memory for Iso-Security-16 (TS = 16-bit, TG = 32B). On an L2 cache miss, we issue a parallel lock tag lookup request to the carve-out memory, which is then cached in the L2 for potential future requests. Both data and carve-out requests are satisfied by the L2 cache before a response is sent to the L1 cache. We assume that the L1 and L2 caches have additional storage for per-cacheline lock tags and that the datapaths from the L2 to the L1 are widened to transfer the tags alongside the data. We evaluate 193 application traces from various domains, including MLPerf [38, 52], high performance computing and sparse linear algebra workloads (*e.g.*, [30, 34, 54, 57, 69, 72] among others), and the STREAM microbenchmark.

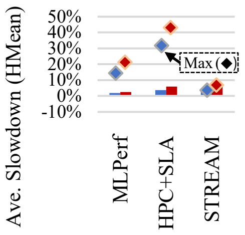
We analyze the reliability of ECC stealing and IMT using software decoders. Random data corruption is estimated using 1e8 randomly-generated errors; all other error patterns are exhaustively evaluated. We estimate the hardware overheads of AFT-ECC using Verilog designs synthesized by the Synopsys toolchain [66] with a 16nm industrial technology library. Hardware area and delay are estimated through standard-cell synthesis and static timing analysis.

Memory tagging security guarantees depend on the number of unique tags and the algorithm used to retag objects [6, 50]. Implicit Memory Tagging is allocator-agnostic, and we evaluate security using the retagging schemes of two existing ARM MTE memory allocators. The GNU C Library (glibc) allocator [17] assigns purely random tags to each memory allocation [10]. On the other hand, Scudo [36], the default system allocator in Android 11, is further constrained to assign random odd or even tags based on the order of memory objects, so that adjacent objects always receive different tags [63]. The ISA instructions that tag and clear memory may reserve tags for other uses; this evaluation assumes that two tags are reserved, as with SPARC ADI [35].

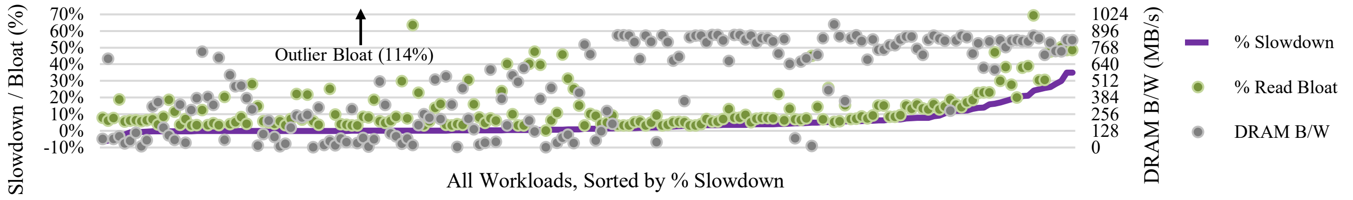
[5] We denote these same-security configurations as iso-security. For instance, iso-security-10 has the same security as IMT-10.



(a) Slowdown Across Workloads



(b) Average/Max



(c) Analysis of the Low-Tag-Storage Carve-Out Slowdowns (Across All Workloads)

Figure 8: Simulation results showing the performance overheads of tag carve-out based memory tagging.

5.2 Performance Evaluation

Figure 8 shows the performance impact of the tag carve-out baselines over 193 different workloads. The tag carve-out can reduce the performance of some workloads by requiring redundant tag memory requests and additional pressure on on-chip caches to store and access the cached tags. Performance overheads depend on the required tag storage; we show a low-tag-storage option representing the ARM MTE (TS=4b, TG=16B) or Iso-Security-10 (TS=8b, TG=32B) configurations, and a high-tag-storage option for Iso-Security-16 (TS=16b, TG=32B). Overheads are modest, on average, and they resemble those reported for the recent GPU tag carve-out implementation LAK [75]. The harmonic mean slowdown across MLPerf workloads is 1.8/2.3% (low-tag-storage/high-tag-storage). Similarly, the HPC and Sparse Linear Algebra workloads suffer 3.7/5.8% average slowdowns, and there are 3.5/6.8% average slowdowns for the STREAM microbenchmarks. Overheads are larger for memory-bandwidth-bound programs and those with fine-grained random access patterns, with maximum slowdowns of 14/21% for MLPerf, 32/43% for HPC+SLA, and 4/7% for STREAM.

Figure 8c further analyses the low-tag-storage slowdowns by also plotting the percent read bloat due to memory tagging, and the average DRAM bandwidth used by the program. Slowdowns grow with either increasing read bloat or for bandwidth-constrained programs, and especially if both are present. The most severe slowdowns are in HPC workloads `LAAMPS` [57] and `AMBER18` [34], which use many fine-grained memory accesses and also have high memory bandwidth demands.

5.3 Reliability Evaluation

Figure 9 shows the SDC probability for different error-detecting codes (R=1–8 bits) a SEC code (R=9 bits), and SEC-DED codes (R=10–16 bits). Two error patterns are shown: (1) random corruption, and (2) 3-bit errors, which have disproportionate SDC risk for SEC-DED codes. These results are collected by selecting a random minimum-weight all-odd-column code for each of the SEC-DED ECCs, and by selecting a random H matrix for the SEC code.[6] Both error-detecting and error-correcting codes suffer from a ~2× higher silent data corruption rate against random corruption for each bit lost. The behavior of 3-bit errors is slightly more complicated, but it is still near a 2× reduction.[7] Intuitively, this leads to the reliability penalty of ECC stealing—for example, if we steal 4-bits of the R = 16 GPU ECC to perform memory tagging (like the SPARC ADI configuration), the SDC risk is amplified by 15.8×. Supporting a large tag through ECC stealing sacrifices error correction, which is highly undesirable due to availability concerns, and it also results in significant reliability degradations. Both Iso-Security-10 and Iso-Security-16 steal enough bits to only leave a single bit for parity. This results in a 1.92×/120× higher SDC risk than IMT for Iso-Security-10 and Iso-Security-16, respectively.

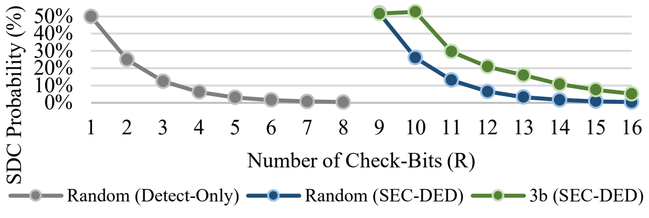


Figure 9: The SDC probability of error-detecting and correcting codes with varying redundancies.

Table 2 shows the IMT behavior using different error patterns. AFT-ECC does not change the reliability characteristics of the underlying SEC-DED ECC code. As expected, both the 10 and 16-bit SEC-DED codes maintain correction of 1-bit data errors and they detect all 2-bit data errors. SEC-DED codes tend to have a greater SDC

[6]There is no all-odd-weight H matrix possible for the SEC code (R=9).
[7]One exception is when going from R = 10 to R = 9, because at R = 9 only error correction is possible. SEC codes do not suffer from the same odd-weight error issues as SEC-DED codes, and thus the 3-bit error probability of the SEC code is slightly better than that of the R = 10 SEC-DED code.

Table 2: The Per-Error-Pattern Behavior of AFT-ECC. CE/DE = Corrected/Detected Error; SDC = Silent Data Corrupt.

	IMT-10			IMT-16		
Redundancy	10b			16b		
Tag Size (TS)	9b			15b		
Pattern	CE	DE	SDC	CE	DE	SDC
Tag Corrupt	0%	100%	0%	0%	100%	0%
1b Data	100%	0%	0%	100%	0%	0%
2b Data	0%	100%	0%	0%	100%	0%
3b Data	0%	47.53%	52.47%	0%	95.05%	4.952%
4b Data	0%	99.80%	0.001995%	0%	99.98%	0.0001841%
Rand. Data*	0%	73.92%	25.98%	0%	99.58%	0.4154%

*Also equivalent to a simultaneous tag mismatch and data error.

risk for odd-weight data errors than even-weight ones, as is evident from the DE and SDC probabilities. IMT detects all corrupted tags due to its alias-free tag submatrix. Even-weight $\geq$ 2b data errors are misattributed as TMMs (unless precise driver-side diagnosis is performed),[8] while odd-weight multi-bit errors are reported as DUEs. In the rare event of a simultaneous tag mismatch and data error, the behavior is the same as with random data corruption.

5.4 Security Evaluation

Memory tagging security increases with the number of unique valid tags, with a detection rate of $100\% - \frac{100\%}{\text{Num.Tags}}$. IMT allows for large tags to be used without efficiency or resilience concerns—IMT-10 supports a 9-bit tag, and IMT-16 a 15-bit tag. These large tags result in a 36× (IMT-10) or 2340× (IMT-16) lower misdetection probability than that of ARM MTE and SPARC ADI with 4-bit tags.

The precise memory tagging security guarantees depend on the algorithm used to retag objects. Implicit Memory Tagging provides 98.804% (IMT-10) or 99.997% (IMT-16) detection of memory safety violations using the retagging scheme of the glibc allocator [10, 17], up from 92.857% for the industry baselines. The Android Scudo allocator [36] is designed with slightly different tradeoffs. It assigns alternating odd and even tags to adjacent objects, leading to a 100% detection rate against adjacent buffer overflows.[9] However, this comes at a 2× misdetection penalty against non-adjacent buffer overflows. IMT-10 and IMT-16 still provide strong probabilistic security guarantees of 99.608% and 99.994%, respectively. The industry baselines suffer from worse absolute security degradation, only providing 85.714% detection using the Scudo memory allocator.

5.5 Hardware Overheads

Figure 10 shows the SEC-DED decoder hardware. The decoder generates a syndrome by XORing the encoded data with the received check-bits. For single-bit errors, the matching H column bit is corrected, while a DUE is raised if no match is found. IMT/AFT-ECC widen the encoder with TS new columns and introduce additional

[8]This behavior is because IMT uses the maximum tag size for 32B data. With a smaller-than-maximum tag, IMT would have < 100% TMMs for even-weight errors—each bit of TS reduction would decrease the even-weight-error misattribution risk by ~2×.

[9]This 100% security guarantee assumes that the attacker cannot change the key tag bits. If they can, the approach provides the same probabilistic detection as against non-adjacent buffer overflows.

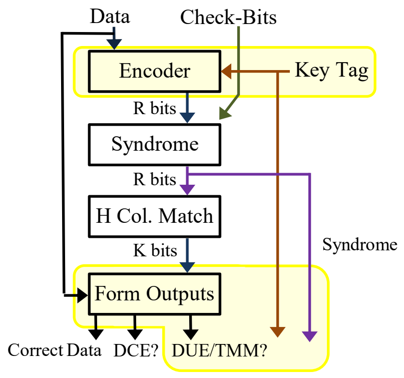


Figure 10: The hardware for a SEC-DED decoder with the minor changes needed for IMT/AFT-ECC highlighted.

outputs to the decoder. Table 3 reveals minimal hardware overheads—the hardware additions correspond to < 200 added AND2 gates worth of area per encoder and < 400 per decoder. IMT/AFT-ECC add no delay, as T_{TS} introduces only two bits per row, which does not force another level in the encoder's XOR tree structure.

6 OTHER RELATED WORK

GPUShield [33] uses the GPU compiler and driver to craft a per-kernel allocation bounds table that is indexed by hardware using the upper pointer bits to validate each memory access. Although both IMT and GPUShield utilize upper pointer bits for memory safety, their approaches differ and IMT offers distinct advantages. IMT avoids extensive compiler modifications and it can protect pre-compiled third-party libraries without source code access. IMT does not require a new on-chip cache hierarchy or substantial hardware modifications, and it offers temporal as well as spatial safety guarantees. IMT allows unlimited allocations, while GPUShield is restricted by table size—it only gives a single table entry to the device-side heap (for dynamic per-thread allocations [46]) for this reason. While the performance overheads of GPUShield are low, those of IMT are zero and there is a quantitative benefit for some programs. We simulate our workloads with a tagged base-and-bounds approach somewhat similar to GPUShield (with no static bounds checking), and find that 40 of the workloads suffer from > 0.5% slowdown. Among these workloads, there is a harmonic mean slowdown of 0.96% and a maximum slowdown of 14%.

Previous research [26, 50] employs tree-like structures to increase common-case tag cache hit rates [26] or enlarge average tag sizes [50]. IMT supports large tags without extra storage, while avoiding the performance overheads and design complexity of tree traversal on memory accesses.

LAK [75] proposes a performant tag carve-out organization for GPUs. Our tag carve-out baseline is similar to LAK, and we observe similar performance with memory-intensive workloads. IMT has storage and performance benefits over such tag carve-out designs.

SafeMem [51] uses ECC poisoning to provide trip-wire-based memory safety. Like IMT, it leverages ECC error detection to function with little-to-no performance overheads. However, it lacks protection against non-adjacent buffer overflows.

Table 3: Hardware Overheads of IMT/AFT-ECC.

Unit	Area (AND2 / +%)	Delay (ns / +%)
Encoders		
SEC-DED Encoder (10b)	2406 / -	0.12 / -
AFT-ECC Encoder (10b)	2559 / +6.48%	0.12 / -
SEC-DED Encoder (16b)	1483 / -	0.10 / -
AFT-ECC Encoder (16b)	1639 / +10.5%	0.10 / -
Decoders		
SEC-DED Decoder (10b)	4109 / -	0.23 / -
AFT-ECC Decoder (10b)	4296 / +4.56%	0.23 / -
SEC-DED Decoder (16b)	4582 / -	0.22 / -
AFT-ECC Decoder (16b)	4967 / +8.41%	0.22 / -

MUSE ECC [37] uses residue-code error correction to chipkill-protect DDR4 and DDR5 memories with less redundancy than Reed-Solomon codes. They then use the leftover ECC storage for memory tags, at the expense of error detection coverage. This approach is akin to ECC stealing, since each bit of saved redundancy could also be used to reduce the SDC risk of MUSE ECC by ~2×. If AFT-ECC were sufficiently modified to work with MUSE ECC, it could be used to implicitly store the tag bits while using the full redundancy for superior error detection.

7 DISCUSSION & FUTURE WORK

7.1 Support for Other ECCs

This paper considers SEC-DED ECC, which is purportedly used for GPU DRAM, caches, and register files [1, 42]. Structured large-granularity errors are also important—for instance, neutron beam testing experiments report byte errors to be the most common multi-bit error in modern DRAM [64], and burst errors as the most common multi-bit error pattern in SRAM [41]. Thus, an extension of Alias-Free Tagged ECC to other ECC codes (such as symbol-based ECC or burst-correcting ECC) is an area of future research. It is trivial to support a TS within the correction capabilities of a shortened code, but support for a large maximum-length tag should also be possible with proper code design. Analysis of tag size bounds and AFT-ECC construction procedures with other ECC codes is left for future work.

7.2 IMT on Other Platforms

IMT can fundamentally be applied to CPUs or other accelerators with an ECC-protected memory hierarchy, but some challenges must be carefully addressed during adoption. CPUs often use symbol-based chipkill ECC [27] to tolerate an entire failing DRAM device; this is an area of future AFT-ECC work. Using IMT alongside hardware memory prefetchers will require careful organization. Some CPUs protect the write-through L1 cache with parity only and rely on L2 data replication for correction [65, 68], unlike the exclusive GPU shared memory scratchpad that requires error correction. Hence, extra L1 cache redundancy may be needed for IMT on CPUs. Finally, small allocations are relatively more common on CPUs than GPUs, and ECC codewords are often at a larger 64B cacheline granularity—this could lead to significant memory fragmentation from tagging at the cacheline granularity. Because of such differences, the detailed evaluation of IMT on other computing platforms is left for future work.

7.3 IMT with Improved Memory Allocators

Implicit Memory Tagging optimizes the storage, movement, and checking of the lock tag, which are the primary sources of memory tagging overheads. Accordingly, this paper does not cover improvements to the memory tagging algorithm, itself. IMT allows for efficient implementation of a large tag size, which could enable further flexibility in the memory allocator design. For instance, a modified allocator might guarantee deterministic detection up to a certain number of live allocations, or guarantee use-after-free detection until a memory location is reallocated a certain number of times, without prohibitive detection degradation for non-adjacent buffer overflows. Improved memory allocators that leverage the large IMT tag is an area of future innovation.

7.4 Other Uses for Alias-Free Tagged ECC

Tags for Low-Cost DRAM Caches: Alias-Free Tagged ECC could perform DRAM cache tag checking alongside regular DRAM reads. The ability to associate a tag with each 32B memory location would allow for fine-grained DRAM cachelines without any tag storage overheads. This could be useful for large-footprint workloads with random accesses, such as graph analytics. However, this technique may have some limitations due to the AFT-ECC constraints. For example, only a write-through DRAM cache [61] might be supported (without precise TMM diagnosis), since a dirty write-back tag cannot be safely extracted from an AFT-ECC codeword without risking SDC for misattributed multi-bit errors.

Bulk Cache Invalidation: Bulk invalidation is used in the GPU L1 for software coherence. One downside of bulk invalidation is the latency of the operation, especially as cache sizes get large (or DRAM caches are used). Alias-Free Tagged ECC could be used to implement an invalidation-epoch-counting tag that considers tag mismatches to be invalid cache entries. This reduces the need to crawl the cache to once every 2^{TS} bulk invalidations. A similar idea requiring additional cache meta-data is described in CARVE [74]. AFT-ECC could achieve the same behavior without cache storage.

8 CONCLUSION

This paper describes a novel class of error codes and an application of these codes for no-overhead memory tagging. Alias-Free Tagged ECC (AFT-ECC) embeds a meta-data tag in the ECC check-bits, unambiguously identifying tag mismatches while preserving single-bit data error correction. Implicit Memory Tagging (IMT) uses Alias-Free Tagged ECC to check for the equivalence of a large memory tag in addition to detecting and correcting data errors. We apply IMT to GPUs, and demonstrate that it outperforms alternative hardware-accelerated memory tagging approaches in performance, security, and reliability.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We appreciate the paper discussions we have had with Adam Zabrocki, Nirmal Saxena, Mark Hairgrove, and Siddhartha Chhabra at NVIDIA.

REFERENCES

[1] Advanced Micro Devices (AMD), Inc. 2012. *AMD Graphics Cores Next (GCN) Architecture*. Advanced Micro Devices (AMD), Inc. https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf

[2] Advanced Micro Devices (AMD), Inc. 2015. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 30h-3Fh Processors*. Advanced Micro Devices (AMD), Inc. Publication #49125 Rev 3.06.

[3] Advanced Micro Devices (AMD), Inc. 2022. AMD EPYC 9004 Series Architecture Overview. https://www.amd.com/system/files/documents/58015-epyc-9004-tg-architecture-overview.pdf Publication #58015.

[4] Arm Ltd. 2021. *Armv8.5-A Memory Tagging Extension*. Arm Ltd. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

[5] Arm Ltd. 2022. *Arm Architecture Reference Manual: For A-Profile Architecture*. Arm Ltd. https://developer.arm.com/documentation/ddi0487/ia Version I.A, pp. 5219–5230.

[6] Joe Bialek, Ken Johnson, Matt Miller, and Tony Chen. 2020. Security Analysis of Memory Tagging. https://github.com/microsoft/MSRC-Security-Research/raw/master/papers/2020/Security%20analysis%20of%20memory%20tagging.pdf

[7] Esha Choukse, Michael B. Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W. Keckler. 2020. Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 926–939.

[8] Bang Di, Jianhua Sun, and Hao Chen. 2016. A Study of Overflow Vulnerabilities on GPUs. In *Proceedings of the International Conference on Network and Parallel Computing (NPC)*. 103–115.

[9] Bang Di, Jianhua Sun, Hao Chen, and Dong Li. 2021. Efficient Buffer Overflow Detection on GPU. *IEEE Transactions on Parallel Distributed Systems* 32, 5 (2021), 1161–1177. https://doi.org/10.1109/TPDS.2020.3042965

[10] Richard Earnshaw. 2020. Add AARCH64-Specific Files for Memory Tagging Support. https://sourceware.org/git/?p=glibc.git;a=commit;h=d27f0e5d889f4bf4a796fe2a883b2f264bf40c12

[11] Christopher Erb and Joseph L. Greathouse. 2018. ClARMOR: A Dynamic Buffer Overflow Detector for OpenCL Kernels. In *Proceedings of the International Workshop on OpenCL (IWOCL)*. 1–2.

[12] Eiji Fujiwara. 2006. *Code Design for Dependable Systems: Theory and Practical Application*. Wiley-Interscience.

[13] Richard Henry Gumpertz. 1981. *Error Detection with Memory Tags*. Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA.

[14] Richard Henry Gumpertz. 1983. Combining Tags with Error Codes. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 160–165.

[15] M.Y. Hsiao. 1970. A Class of Optimal Minimum Odd-Weight-Column SEC-DED Codes. *IBM Journal of Research and Development* 14, 4 (1970), 395–401.

[16] John Hubbard. 2017. Using HMM to Blur the Lines between CPU and GPU Programming. GPU Technology Conference (GTC). http://on-demand.gputechconf.com/gtc/2017/presentation/s7764_john-hubbardgpus-using-hmm-blur-the-lines-between-cpu-and-gpu.pdf

[17] Free Software Foundation Inc. 2023. The GNU C Library Reference Manual, for version 2.37. https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html

[18] Intel Corp. 2017. 5-Level Paging and 5-Level EPT. https://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html Document Number: 335252-002, Revision 1.1.

[19] JEDEC Solid State Technology Association 2013. *Graphics Double Data Rate (GDDR5) SGRAM Standard, JESD212B*. JEDEC Solid State Technology Association.

[20] JEDEC Solid State Technology Association 2015. *Graphics Double Data Rate (GDDR5X) SGRAM Standard, JESD232*. JEDEC Solid State Technology Association.

[21] JEDEC Solid State Technology Association 2015. *High Bandwidth Memory (HBM) DRAM, JESD256A*. JEDEC Solid State Technology Association.

[22] JEDEC Solid State Technology Association 2021. *Graphics Double Data Rate 6 (GDDR6) SGRAM Standard, JESD250C*. JEDEC Solid State Technology Association.

[23] JEDEC Solid State Technology Association 2022. *High Bandwidth Memory DRAM (HBM3), JESD238*. JEDEC Solid State Technology Association.

[24] Samuel Jero, Nathan Burow, Bryan Ward, Richard Skowyra, Roger Khazan, Howard Shrobe, and Hamed Okhravi. 2022. TAG: Tagged Architecture Guide. *Comput. Surveys* (May 2022).

[25] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv preprint arXiv:1804.06826* (April 2018), 17–24.

[26] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N.M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. 2017. Efficient Tagged Memory. In *Proceedings of the International Conference on Computer Design (ICCD)*. 641–648.

[27] Jungrae Kim, Michael B Sullivan, and Mattan Erez. 2015. Bamboo ECC: Strong, Safe, and Flexible Codes for Reliable Computer Memory. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 101–112.

[28] Jungrae Kim, Michael B. Sullivan, Sangkug Lym, and Mattan Erez. 2016. All-Inclusive ECC: Thorough End-to-End Protection for Reliable Computer Memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 622–633.

[29] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 361–372.

[30] KTH Royal Institute of Technology. 2023. GROMACS. NVIDIA GPU Cloud. https://catalog.ngc.nvidia.com/orgs/hpc/containers/gromacs

[31] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. 2013. Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-Based Security. In *Proceedings of the SIGSAC Conference on Computer and Communications Security*. 721–732.

[32] Hugo Landau. 2022. The Talos II, Blackbird POWER9 Systems Support Tagged Memory. https://www.devever.net/~hl/power9tags

[33] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing GPU via Region-Based Bounds Checking. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 27–41.

[34] Tai-Sung Lee, David S Cerutti, Dan Mermelstein, Charles Lin, Scott LeGrand, Timothy J Giese, Adrian Roitberg, David A Case, Ross C Walker, and Darrin M York. 2018. GPU-Accelerated Molecular Dynamics and Free Energy Methods in Amber18: Performance Enhancements and New Features. *Journal of Chemical Information and Modeling* 58, 10 (2018), 2043–2050.

[35] Linux Kernel Organization 2019. *Application Data Integrity (ADI)*. Linux Kernel Organization. https://www.kernel.org/doc/Documentation/sparc/adi.rst

[36] LLVM Project. 2023. Scudo Hardened Allocator. https://llvm.org/docs/ScudoHardenedAllocator.html

[37] Evgeny Manzhosov, Adam Hastings, Meghna Pancholi, Ryan Piersma, Mohamed Tarek Ibn Ziad, and Simha Sethumadhavan. 2022. Revisiting Residue Codes for Modern Memories. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 73–90.

[38] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. 2020. MLPerf Training Benchmark. *Proceedings of Machine Learning and Systems* 2 (2020), 336–349.

[39] Andrea Miele. 2016. Buffer Overflow Vulnerabilities in CUDA: A Preliminary Analysis. *Journal of Computer Virology and Hacking Techniques* 12, 2 (2016), 113–120.

[40] M Miller. 2019. Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape. Blue Hat IL. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[41] A. Neale and M. Sachdev. 2016. Neutron Radiation Induced Soft Error Rates for an Adjacent-ECC Protected SRAM in 28nm CMOS. *IEEE Transactions on Nuclear Science* 63, 3 (June 2016), 1912–1917.

[42] NVIDIA Corp. 2016. *NVIDIA Tesla P100—The Most Advanced Data Center Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU*. NVIDIA Corp. http://www.nvidia.com/object/pascal-architecture-whitepaper.html

[43] NVIDIA Corp. 2019. *NVIDIA T4 Tensor Core GPU*. NVIDIA Corp. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf

[44] NVIDIA Corp. 2021. *NVIDIA A100 GPU Memory Error Management*. NVIDIA Corp. https://docs.nvidia.com/deploy/a100-gpu-mem-error-mgmt/index.html

[45] NVIDIA Corp. 2022. *Dynamic Page Retirement*. NVIDIA Corp. https://docs.nvidia.com/deploy/dynamic-page-retirement/index.html

[46] NVIDIA Corp. 2023. CUDA C Programming Guide, Section 10.34: Dynamic Global Memory Allocation and Operations. NVIDIA Developer Zone. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf Release 12.1, pages 228–231.

[47] Oracle Corp. 2015. *adi_memset Documentation*. Oracle Corp. https://docs.oracle.com/cd/E86824_01/html/E54766/adi-memset-3c.html

[48] Oracle Corp. 2015. *Hardware-Assisted Checking Using Silicon Secured Memory (SSM)*. Oracle Corp. https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html

[49] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. 2021. Mind Control Attack: Undermining Deep Learning With GPU Memory Exploitation. *Computers & Security* 102 (2021), 102–115.

[50] Aditi Partap and Dan Boneh. 2022. Memory Tagging: A Memory Efficient Design. *arXiv preprint arXiv:2209.00307* (Nov. 2022), 1–16.

[51] Feng Qin, Shan Lu, and Yuanyuan Zhou. 2005. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 291–302.

[52] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. MLPerf Inference Benchmark. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 446–459.

[53] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. 2013. A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 86–98.

[54] Nikolay Sakharnykh. 2016. High-Performance Geometric Multi-Grid with GPU Acceleration. https://developer.nvidia.com/blog/high-performance-geometric-multi-grid-gpu-acceleration/

[55] Nikolay Sakharnykh. 2017. Unified Memory on Pascal and Volta. GPU Technology Conference (GTC). http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf

[56] Nikolay Sakharnykh. 2019. Memory Management on Modern GPU Architectures. GPU Technology Conference (GTC). https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9727-memory-management-on-modern-gpu-architectures.pdf

[57] Sandia National Lab. 2023. LAAMPS. NVIDIA GPU Cloud. https://catalog.ngc.nvidia.com/orgs/hpc/containers/lammps

[58] N. Saxena, Chien Chen, R. Swami, H. Osone, S. Thusoo, D. Lyon, D. Chang, A. Dharmaraj, N. Patkar, Y. Lu, and B. Chia. 1995. Error Detection and Handling in a Superscalar, Speculative Out-of-Order Execution Processor System. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*. 464–471.

[59] Yiannakis Sazeides, Emre Özer, Danny Kershaw, Panagiota Nikolaou, Marios Kleanthous, and Jaume Abella. 2013. Implicit-Storing and Redundant-Encoding-of-Attribute Information in Error-Correction-Codes. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 160–171.

[60] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and How It Improves C/C++ Memory Safety. *arXiv preprint arXiv:1802.09517* (Feb. 2018), 1–14.

[61] Jaewoong Sim, Gabriel H Loh, Hyesoon Kim, Mike OConnor, and Mithuna Thottethodi. 2012. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 247–257.

[62] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *Proceedings of the Symposium on Security and Privacy (SP)*. 1275–1295.

[63] Evgenii Stepanov, Kostya Serebryany, Peter Collingbourne, Mitch Phillips, and Vitaly Buka. 2020. Memory Tagging in LLVM and Android. In *LLVM Developer Meeting*. https://llvm.org/devmtg/2020-09/slides/Stepanov-Memory_tagging_in_LLVM_and_Android.pdf

[64] Michael B. Sullivan, Nirmal Saxena, Mike O'Connor, Donghyuk Lee, Paul Racunas, Saurabh Hukerikar, Timothy Tsai, Siva Kumar Sastry Hari, and Stephen W. Keckler. 2021. Characterizing and Mitigating Soft Errors in GPU DRAM. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 641–653.

[65] Sun Microsystems 2008. *OpenSPARC T2 System-on-Chip (SoC) Microarchitecture Specification*. Sun Microsystems.

[66] Synopsys Inc. 2022. Design Compiler T-2022.03-SP5.

[67] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the Symposium on Security and Privacy (SP)*. 48–62.

[68] Joel M Tendler, J Steve Dodson, JS Fields, Hung Le, and Balaram Sinharoy. 2002. POWER4 System Microarchitecture. *IBM Journal of Research and Development* 46, 1 (2002), 5–25.

[69] University of Illinois Urbana-Champaign. 2023. NAMD. NVIDIA GPU Cloud. https://catalog.ngc.nvidia.com/orgs/hpc/containers/namd

[70] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *Proceeding of the Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. 86–106.

[71] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. 2021. Need for Speed: Experiences Building a Trustworthy System-Level GPU Simulator. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 868–880.

[72] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 1–12.

[73] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the USENIX Security Symposium*. 719–732.

[74] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. 2018. Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 339–351.

[75] Chaochao Zhang and Rui Hou. 2022. LAK: A Low-Overhead Lock-and-Key Based Schema for GPU Memory Safety. In *Proceedings of the International Conference on Computer Design (ICCD)*. 705–713.

[76] Benjamin Zorn, Paul Hilfinger, Kinson Ho, and James Larus. 1987. *SPUR Lisp*. Technical Report. https://www2.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-373.pdf