Reliability-Aware Garbage Collection for Hybrid HBM-DRAM Memories

WENJIE LIU, Ghent University
SHOAIB AKRAM, Australian National University
JENNIFER B. SARTOR, Ghent University
LIEVEN EECKHOUT, Ghent University

Emerging workloads in cloud and data center infrastructures demand high main memory bandwidth and capacity. Unfortunately, DRAM alone is unable to satisfy contemporary main memory demands. High-bandwidth memory (HBM) uses 3D die-stacking to deliver 4–8× higher bandwidth. HBM has two drawbacks: (1) capacity is low, and (2) soft error rate is high. Hybrid memory combines DRAM and HBM to promise low fault rates, high bandwidth, and high capacity. Prior OS approaches manage HBM by mapping pages to HBM versus DRAM based on hotness (access frequency) and risk (susceptibility to soft errors). Unfortunately, these approaches operate at a coarse-grained page granularity, and frequent page migrations hurt performance.

This paper proposes a new class of reliability-aware garbage collectors for hybrid HBM-DRAM systems which place hot and low-risk objects in HBM and the rest in DRAM. Our analysis of 9 real-world Java workloads shows that: (1) newly-allocated objects in the nursery are frequently written, making them both hot and low-risk, (2) a small fraction of the mature objects are hot and low-risk, and (3) allocation site is a good predictor for hotness and risk. We propose RiskRelief, a novel reliability-aware garbage collector that uses allocation site prediction to place hot and low-risk objects in HBM. Allocation sites are profiled offline and RiskRelief uses heuristics to classify allocation sites as DRAM and HBM. The proposed heuristics expose Pareto-optimal trade-offs between soft error rate (SER) and execution time. RiskRelief improves SER by 9× compared to an HBM-Only system while at the same time improving performance by 29% compared to a DRAM-Only system. Compared to a state-of-the-art OS approach for reliability-aware data placement, RiskRelief eliminates all page migration overheads, which substantially improves performance while delivering similar SER. Reliability-aware garbage collection opens up a new opportunity to manage emerging HBM-DRAM memories at fine granularity while requiring no extra hardware support and leaving the programming model unchanged.

CCS Concepts: • **Computer systems organization → Reliability**; **Heterogeneous (hybrid) systems**; **Processors and memory architectures**; • **Software and its engineering → Garbage collection**.

Additional Key Words and Phrases: Soft-error reliability, hybrid memories, high-bandwidth memory, garbage collection

This work is supported by the European Research Council (ERC) Advanced Grant agreement no. 741097, and FWO projects G.0434.16N and G.0144.17N. Wenjie Liu is supported through a CSC fellowship.

Authors' addresses: L. Liu, J. B. Sartor and L. Eeckhout, ELIS Department, Ghent University, iGent, Technologiepark 126, 9052 Zwijnaarde, Belgium; emails: Wenjie.Liu@UGent.be, Jennifer.Sartor@UGent.be, Lieven.Eeckhout@UGent.be; S. Akram, Research School of Computer Science, Australian National University, 108 North Road Canberra ACT 2600, Australia; email: shoaib.akram@anu.edu.au.

XXXX-XXXX/2020/1-ART1
https://doi.org/10.1145/3431803

ACM Reference Format:
Wenjie Liu, Shoaib Akram, Jennifer B. Sartor, and Lieven Eeckhout. 2020. Reliability-Aware Garbage Collection for Hybrid HBM-DRAM Memories. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (January 2020), 25 pages. https://doi.org/10.1145/3431803

1 INTRODUCTION

Emerging cloud workloads, such as machine learning inference and stream analytics, have encouraged new throughput-oriented compute platforms. These platforms consist of many-core processors, graphic processing units, and a range of accelerators. Altogether, these compute platforms have an insatiable demand for main memory bandwidth. The confluence of ever-growing compute power and the slow historical growth in pin count for off-chip communication [33] has exacerbated the memory bandwidth wall [56]. High Bandwidth Memory (HBM) [7], i.e., 3D-stacked DRAM, delivers higher bandwidth than traditional DRAM, while consuming less power and space [10, 18, 19, 22, 28, 38, 39, 61].

HBM has two shortcomings though: (1) capacity is limited to a couple GBs, and (2) soft error rate is high due to higher density and new failure modes [34, 49]. Hybrid HBM-DRAM memory combines the best of both worlds to provide high capacity and high bandwidth. Unfortunately, unless properly managed, HBM reliability is a concern. Our experimental results reveal that an HBM-Only system yields 34% higher performance than a DRAM-Only system, but the entire program heap is capacity-limited and, moreover, is highly vulnerable to soft errors. A DRAM-Only system, on the other hand, is substantially more reliable (by at least two orders of magnitude), but at the expense of considerably lower performance compared to HBM-Only. The goal of this work is to achieve the best of both worlds, i.e., deliver high reliability while achieving high performance.

A flurry of prior work proposes hardware and OS approaches to optimize hybrid memory performance. Specifically, hardware approaches use HBM as a cache for DRAM [18, 19, 37, 41, 43], whereas OS approaches map frequently accessed pages in HBM [51, 54, 55, 62]. Only recently have researchers turned attention to data placement approaches to address the low reliability of HBM [27]. Indeed, soft error rates in production systems are continuously increasing, and they grow proportionally with information density [42]. Hardware-only approaches to tackle reliability are insufficient because they will soon require impractical error detection and correction capabilities [47]. OS approaches [27] also face drawbacks: (1) they are reactive, (2) page migrations incur significant performance penalty, and (3) they are coarse-grained and require excessive HBM capacity.

This paper takes a different, so far unexplored, approach by leveraging garbage collection in modern managed languages to place program data in hybrid HBM-DRAM memory at a finer granularity than state-of-the-art OS approaches. Garbage collection (GC) in managed languages such as Java, C#, JavaScript, Python, and Ruby manages virtual heap memory on behalf of the programmer. Most high-performance GCs place newly allocated (young) objects in a small nursery space. A nursery collection copies surviving objects to the mature space. This generational heap organization leads to short pause times and high application (mutator) locality and performance [8]. Our analysis of various Java applications from the DaCapo suite [13] shows that: (1) nursery objects are hot (frequently accessed) and low-risk (highly mutated), and (2) only a small fraction of nursery survivors are hot and low-risk. These results reveal an opportunity to effectively manage HBM-DRAM memory.

This work proposes a new class of *reliability-aware garbage collectors* for hybrid memory. These collectors place hot and low-risk objects in HBM to improve reliability and performance. The remaining objects are placed in DRAM to utilize its large capacity. Reliability-aware garbage collection overcomes the disadvantages of the state-of-the-art OS approach. Specifically, prediction

enables pro-active allocation of objects in HBM as opposed to reactive page migrations. Moreover, placing objects using GC eliminates the overhead of costly page migrations.

In this paper, we propose two reliability-aware garbage collectors. RiskRelief-Nursery (RR-N) places the nursery in HBM and the mature space in DRAM. It requires minimal changes to the Java runtime but is highly effective in delivering low soft error rates compared to an HBM-Only system, while improving performance compared to a DRAM-Only system. RiskRelief-Mature (RR-M) places the nursery in HBM and exploits offline program profiling to place hot and low-risk nursery survivors in HBM. We show that mature object hotness and risk are predictable on a per allocation-site basis. Surprisingly perhaps, we find that object hotness and risk are weakly correlated. Hence, placing objects in HBM based solely on hotness significantly hurts reliability. The insight is to place objects in HBM versus DRAM based on hotness *and* risk.

Based on these observations, we propose heuristics to classify allocation sites as DRAM and HBM. Allocation sites are classified as HBM if most objects they allocate are hot and low-risk. All other allocation sites default as DRAM. We generate this per allocation-site advice offline and feed it to RR-M. In turn, RR-M uses the advice during runtime to place nursery survivors in HBM or DRAM. Our proposed heuristics expose previously unseen Pareto-optimal trade-offs between execution time and soft error rate. A single profiling run generates a range of advices for the GC runtime. Thus, depending upon factors such as environmental conditions, available HBM capacity and performance goals, a system operator can adjust the advice fed to RR-M to meet specific constraints.

Our experimental results show that RR-N reduces the overall soft error rate by 18× on average compared to an HBM-Only system, while improving performance over a homogeneous DRAM-Only system by 20%. The state-of-the-art OS solution by Gupta et al. [27] achieves similar SER as RR-N, however, performance is substantially worse (even worse than the DRAM-Only system) because of the high cost of TLB shootdowns on modern x86 multicores [51]. Both RR-N and the prior OS approach use a modest 128 MB of HBM on a 32-core platform. RR-M uses an additional 18% of HBM capacity but delivers 29% higher performance compared to a DRAM-Only system. Higher HBM capacity impacts overall SER and RR-M reduces SER by 9× over HBM-Only.
In summary, the main contributions of this paper are:

- hotness (access frequency) and risk (susceptibility to soft errors) characterization of objects in Java applications, showing that hotness and risk are only weakly correlated;
- showing that allocation site is a good predictor for object hotness and risk;
- the design and implementation of reliability-aware garbage collection for hybrid HBM-DRAM memories to minimize soft error rate while maximizing overall application performance — in contrast, performance-optimized HBM-DRAM management significantly hurts reliability;
- profile-driven RiskRelief reliability-aware collectors that exploit allocation-site prediction to place hot and low-risk objects in HBM and the rest in DRAM;
- a profiling framework to measure object hotness and risk on a per allocation-site basis; two heuristics to generate the allocation advice for GC; and a compilation framework that exploits the advice to steer allocation of objects in HBM and DRAM.
- simulation and real hardware emulation results motivating hybrid HBM-DRAM memory for Java applications, and showing that RiskRelief collectors manage hybrid HBM-DRAM memory significantly better than state-of-the-art OS approaches.

2 EXPLOITING HIGH-BANDWIDTH MEMORY

In this section, we discuss the motivation for HBM, and we describe its distinct performance and reliability characteristics. We also review existing approaches to manage HBM.

2.1 3D-Stacked Memory

Disruptive approaches to mitigate the memory bandwidth wall are needed [56]. The bandwidth between conventional DRAM and the processor is limited by pin count, which increases by roughly 10% every year [33]. However, compute power grows much more rapidly. Furthermore, having enough pins to stream a 1024-bit word every cycle to the processor would require 40 Watt just for memory I/O [48]. High-bandwidth memory vertically stacks DRAM chips in a 3D arrangement to deliver higher bandwidth than conventional DRAM. Through-silicon vias (TSVs) interconnect the vertically stacked chips using wide communication lanes.

Conventional DRAM technology, e.g., DDR4, places two 64-bit words on the data bus every cycle. Several DRAM chips work in tandem to produce the word. For example, 16 ×4 chips each provide 4 bits every cycle to render a 64-bit word. In contrast, the state-of-the-art HBM standard allows up to 12 dies per stack, and each stack has 8 unique 128-bit channels per stack, leading to a much wider, 1024-bit memory interface [35]. Internally, each DRAM chip consists of many banks. A 64-byte cache line is striped across banks in different DRAM chips to maximize parallelism. Hardware employs error correction codes (ECC) to shield against soft errors. Typically, an additional chip provides ECC protection to the data word. Most commonly, DRAM employs single-error correcting, double-error detecting (SECDED) codes.

HBM inherits the failure modes of conventional DRAM because it uses a similar cell technology and array layout. Unfortunately, new failure modes exist in HBM, for instance, due to TSV failures [34]. HBM also exhibits higher bit density increasing susceptibility to soft errors [7, 27, 34, 36]. Furthermore, HBM employs weaker error correction due to cost and complexity constraints [27, 36]. Put together, HBM reliability is a major concern which necessitates hardware and software approaches to mitigate the vulnerability to soft errors in HBM and improve the overall reliability of the memory system.

2.2 Managing HBM in Hardware

Exploiting HBM as a last-level DRAM cache is predominant. In particular, prior work proposes new organizations for DRAM caches [38], intelligent tag placement (for example, co-locating tags with data) [28, 41], new techniques to reduce the bandwidth consumed by cache operations [18, 19], and techniques to enable set associativity in giga-scale DRAM caches [71]. Prior work also attempts to mitigate the performance overhead of DRAM caches for capacity-limited applications [17]. Although transparent to the software stack, DRAM caches have two drawbacks: (1) they limit the available memory capacity, and (2) they require extensive hardware support because conventional SRAM-based cache organizations are suboptimal for DRAM technology. Moreover, none of this prior work considers the low reliability of HBM, thus rendering program data in HBM highly vulnerable to transient faults. Liu et al. [44] propose Binary Star which coordinates the reliability schemes in the 3D DRAM LLC versus main memory to improve the reliability of the overall memory hierarchy. Binary Star achieves high reliability for the overall memory system with limited performance loss, while requiring modifications to both system software and hardware. RiskRelief does not require any hardware changes.

ECC codes are the first line of defense against transient faults. DRAM scaling relies on ECC hardware because smaller DRAM cells are more susceptible to soft errors. Several works study DRAM soft error rates in the field [59, 63, 64]. Weaker ECC backs die-stacked memory due to implementation costs [36] and thus requires soft error mitigation from other sources, e.g., through software, as we discuss next.

2.3 Managing HBM in the OS

Existing OS approaches to manage hybrid HBM-DRAM memory aim at either maximizing performance or balancing performance and reliability. Performance-focused approaches hurt reliability [55], because they place all hot pages in HBM while being agnostic to soft error vulnerability. Gupta et al. [27] propose a dynamic page migration scheme which estimates page hotness and risk using performance counters and which migrates (every 100 ms) cold and high-risk pages to DRAM, and hot and low-risk pages to HBM. In contrast, we estimate hotness and risk at a much finer granularity of objects. Our solution pro-actively places objects in HBM versus DRAM, and does not require dynamic monitoring nor additional performance counter hardware. We compare to the OS page migration approach in this work.

Oskin and Loh [51] propose OS-managed DRAM caches. Their work shows the high cost of page migrations due to TLB shootdowns. They also explore statically partitioning program data in C applications in DRAM and HBM, albeit with negligible benefits. Their proposal does not consider the heterogeneity in reliability in a hybrid HBM-DRAM memory system. We expose both DRAM and HBM to the OS to exploit full memory capacity. Furthermore, this is the first work to expose 3D-stacked memory to garbage collection in the managed runtime for fine-grained object placement.

3 BACKGROUND

Before describing how RiskRelief predicts hotness and risk and leverages these predictions to manage hybrid HBM-DRAM systems, we first provide additional background in soft error reliability and managed runtimes.

3.1 Soft Error Reliability

RiskRelief builds upon two notions, namely hotness and risk. Intuitively, hotness refers to how frequently an object is accessed, whereas risk refers to how susceptible an object is to soft errors. We now define both concepts and focus on risk more because it is a less well-known metric.

Hotness. Hotness is a well-known concept and typically refers to how frequently a particular code segment executes. Analogously, we define the hotness of an object as to how frequently the object is accessed through read or write operations. We define an object's hotness as the sum of reads and writes to the object. Our analysis shows that of all accesses to objects, 54% of the accesses on average are reads, and 46% are writes. The high percentage of writes motivates our hotness criteria as the sum of reads and writes.

Risk. Quantifying the risk of an object in HBM is more involved. We build upon the mechanistic notion of architectural vulnerability factor (AVF) to quantify susceptibility to soft errors. AVF is the probability that a transient fault leads to an observable program error. To compute AVF, Mukherjee et al. [57] categorize all bits in a hardware structure into two types: (1) those necessary for architecturally correct execution (ACE), and (2) the remaining un-ACE bits. A fault in the ACE bits results in an observable program error (assuming the fault evades ECC hardware), and a fault in un-ACE bits has no bearing on program correctness. A bit can be ACE for only a fraction of the total execution time. The AVF of a hardware structure is the fraction of all bits that are in ACE state during each cycle.

Precisely computing AVF of an object requires tracking every read and write operation. Consider an object O, stored at memory location M, is written at time t_1 and read at times t_2 and t_3, after which O is dead from the program's point of view (i.e., no other memory location points to O). O is ACE for $t_3 - t_1$ time units, namely between the write at t_1 and its last read at t_3. In case the object would have been written at times t_2 and t_3, the object would be un-ACE throughout. In other

words, to precisely compute the AVF of an object, one needs to track all reads and all writes, which is too high overhead to do online in the context of a managed runtime.

Instead, we build upon prior work [27] and use proxy metrics that are easier to collect while correlating well with AVF. The proxies considered are the writes to reads ratio (W_r/R_d) and the writes-squared to reads ratio ($W_r{}^2/R_d$). The intuition behind these proxies is that an object that is written a lot is more likely to lead to more un-ACE periods. We use the writes-squared to reads ratio in this work because it places extra emphasis on the absolute number of accesses [27]. Since writes-squared to reads ratio is inversely proportional to AVF, we refer to it as AVF-X. In other words, a high writes-squared to reads ratio (high AVF-X) means low risk, and vice versa. Soft error rate (SER) is defined as the product of a device's failure-in-time (FIT-Rate) and AVF. FIT-Rate is defined as the raw failure rate due to single event faults, and depends on environmental factors and circuit characteristics.

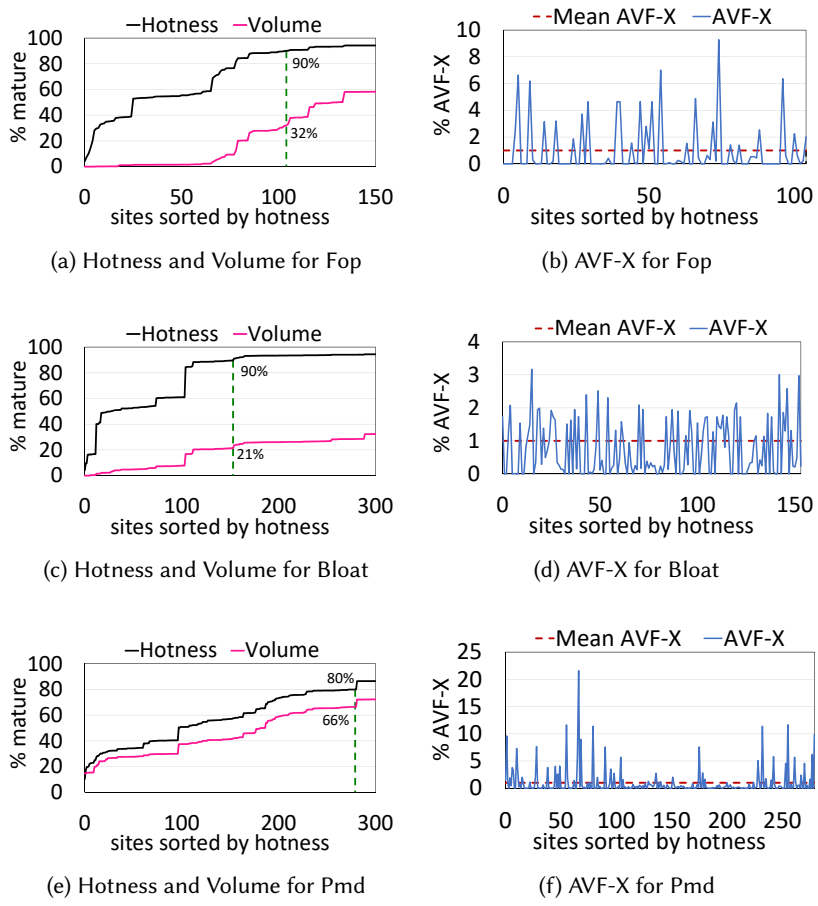
3.2 Managed Runtimes

Java Virtual Machine. This work uses the language runtime to improve system reliability in hybrid memory systems. Our work generalizes to languages with garbage collection, but we use the Java Virtual Machine (JVM) in this work. Exposing HBM to the JVM entails extending the OS NUMA interface [3]. We use the open-source Jikes Research VM (RVM) as our platform. Jikes RVM's modular design makes it easy to modify [5, 6, 12, 25]. Jikes RVM is a meta-circular VM written in Java. It has both a baseline and an optimizing compiler, along with several garbage collectors [11, 14, 60]. The object layout and metadata, and a variety of reference barriers can be changed quickly because of the clean interface between the compiler and garbage collector [25, 69].

Generational Garbage Collection. Despite other differences, garbage collectors in modern languages have converged on a generational heap organization. The generational organization delivers high performance because many objects die young [65]. The application (*mutator*) allocates new objects contiguously into a nursery. When the nursery memory is full, a *minor* collection first identifies live *roots* that point into the nursery, e.g., from global variables, the stack, registers, and the mature space. It then identifies *reachable* objects by tracing references from these roots. It copies reachable objects to a mature space. The nursery space is claimed en masse for fresh allocation.

Nursery size. Nursery size is critical to overall performance, pause time, and space efficiency [8, 11, 66, 72]. A nursery collection incurs a fixed cost to scan the root set and a variable cost depending upon the number of objects that survive a minor collection. Large nurseries sometimes improve performance because objects have more time to die. They, however, increase the overall memory footprint, often unnecessarily retaining dead short-lived objects, and they incur high pause times [50, 72]. We use a 4 MB nursery because prior work establishes that it performs well for our applications [13, 58].

GenImmix. We build on the best-performing collector in Jikes RVM: generational Immix (GenImmix) [14]. We use it as the baseline and modify it to create the RiskRelief collectors. GenImmix uses a copying nursery and a *mark-region* mature space. The mark-region mature space consists of a hierarchy of blocks and lines. Blocks are multiples of page sizes and constitute multiple lines. Lines are multiples of cache line sizes. Objects can span lines but not blocks. Nursery collections copy nursery objects consecutively in space into free lines within blocks in the mature space by incrementing a bump pointer equal to the size of the object. This contiguous allocation outperform free-list allocators due to its locality benefits [11, 14, 32]. Immix reclaims memory at a line and block granularity by marking lines and blocks live when it marks objects live during tracing. To defragment blocks, it combines marking with copying based on runtime heuristics. We use the default settings for the maximum object size (8 KB), for line size (256 bytes), and block size (32 KB).



(a) Hotness and Volume for Fop

(b) AVF-X for Fop

(c) Hotness and Volume for Bloat

(d) AVF-X for Bloat

(e) Hotness and Volume for Pmd

(f) AVF-X for Pmd

Fig. 1. Distribution of hotness and mature heap volume by allocation site (left column), versus risk for the top hottest allocation sites (right column) for Fop (top), Bloat (middle), and Pmd (bottom).

The JVM manages objects larger than an 8 KB threshold separately, allocating them directly into a non-copying large object space [40].

4 HOTNESS AND RISK PREDICTION

This section motivates allocation-site prediction for object hotness and risk.

4.1 Distribution of Hotness and Risk

We start by quantifying hotness and risk across allocation sites for three benchmarks that are representative for the entire benchmark suite, namely Fop, Bloat and Pmd. Figure 1 (left column) shows the cumulative distribution of mature-object hotness and their total volume (as a percentage of total mature allocation) per allocation site. Allocation sites are sorted on the horizontal axis by their hotness. We observe that a large fraction of mature-object accesses are captured by a relatively small fraction of the mature heap. For example, for Fop, 90% of the mature-object accesses are concentrated to only 32% of the mature heap. This result suggests an opportunity to allocate the

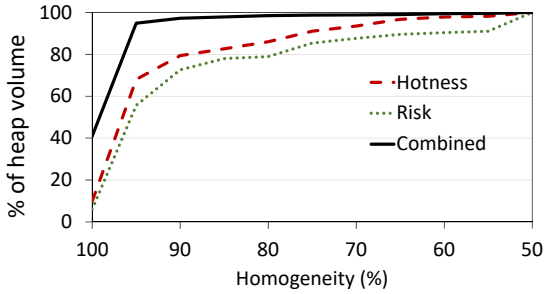


Fig. 2. Percentage heap volume as a function of allocation-site homogeneity for hotness, risk, and combined hotness and risk assuming a 10% cutoff threshold.

relatively small fraction of hot objects in HBM to improve performance while placing the bulk of the mature heap in DRAM to exploit its capacity. Unfortunately, using hotness as the sole criterion to place objects in HBM versus DRAM severely compromises a program's vulnerability to soft errors. The graphs in the right column of Figure 1 report AVF-X for the objects allocated from the top-100 hot allocation sites. To provide a point of reference, we also report mean AVF-X across all mature objects. We observe a remarkable variation in AVF-X across allocation sites from well below to well above the mean. It is clear from these graphs that hotness does not imply low risk, i.e., a hot object may be high-risk or low-risk. In other words, hotness is not predictive for risk. This result implies that using hotness alone as a criterion to classify allocation sites as low- versus high-risk severely compromises soft error vulnerability. Instead, we need a method that classifies allocation sites for both hotness and risk combined, which is what we describe next.

4.2 Allocation-Site Homogeneity

The key insight that underpins RiskRelief is that allocation site is a good predictor for both hotness *and* risk. To demonstrate this is indeed the case, we first compute the hotness and risk for all objects and we determine which objects are among the top 10% (cutoff-threshold) for either criterion. More specifically, we label an object as hot if it is among the 10% hottest objects; if not, the object is classified as cold. Similarly for risk, we label an object as low-risk if it is among the 10% lowest-risk objects; otherwise, the object is classified as high-risk. We then compute for each allocation site, the fraction hot versus cold objects, the fraction low-risk versus high-risk objects, and the fraction of objects that are both hot and low-risk (i.e., combined). We define *homogeneity* of an allocation site with respect to hotness, risk or combined hotness/risk, as the fraction of objects that are classified in the same category. For example for the combined metric, perfect (100%) homogeneity means that all objects allocated from this site are both hot and low-risk, or they are not, i.e., they are either cold or high-risk. On the other hand, a value of 50% means no homogeneity, i.e., 50% of objects are hot and low-risk, whereas the remaining 50% is either cold or high-risk.

Figure 2 reports the percentage heap volume as a function of allocation site homogeneity for hotness, risk, and the combined metric; we report average results across all benchmarks. This graph shows the fraction of heap volume allocated by sites that have a homogeneity of at least N%, with N varying from 100 to 50%. The higher the fraction heap volume covered, the better. As expected, heap volume increases with decreasing allocation site homogeneity. At 100% homogeneity, a relatively small fraction of the total heap volume is covered. However, reducing homogeneity quickly increases the heap volume covered. At 50% homogeneity, the entire heap is covered. The most important, and perhaps surprising, insight from this graph is that the combined metric outperforms the isolated

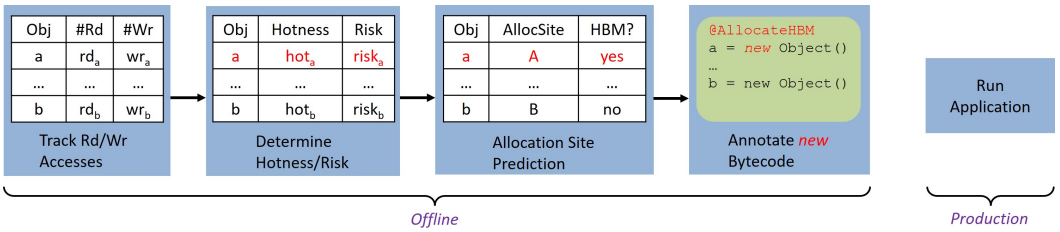


Fig. 3. Overview of RiskRelief. Offline analysis records the number of reads and writes to all objects. Then, per-object hotness and risk metrics are used to generate an allocation site classification advice which serves as input to a bytecode rewriter. The rewriter annotates hot and low-risk sites as HBM, steering the garbage collector to place objects in HBM.

hotness and risk metrics. For example, for 90% homogeneity, more than 97% of the heap is correctly classified for the combined metric, versus 79% and 72% for hotness and risk, respectively. This implies that *allocation site is a more accurate predictor for hotness and risk combined, than for hotness and risk in isolation.* The intuition is that fewer objects satisfy both the hotness and risk thresholds. We thus conclude that allocation site is a very accurate predictor to predict whether objects are hot and low-risk for placement in HBM.

Note that high allocation site homogeneity does not imply that the majority of objects are both hot and low-risk. In fact, an allocation site can have high homogeneity but produce predominantly cold objects, or produce predominantly high-risk objects, or produce predominantly hot and low-risk objects. We only want allocation sites that allocate hot and low-risk objects to be classified as HBM. We find that RiskRelief is sensitive to the object hotness and risk cutoff threshold, but is rather insensitive to the allocation site homogeneity threshold. We use a default object hotness and risk cutoff threshold of 20% and explore its sensitivity in the evaluation section. We use an aggressive allocation site homogeneity threshold of 1% to classify allocation sites as HBM that produce even a small fraction of hot and low-risk objects, i.e., at least 1% of the objects allocated from this site are both hot and low-risk. (Note that because of high allocation site homogeneity, this implies that most objects are hot and low-risk.) We choose this aggressive threshold to make sure that hot and low-risk objects are allocated in HBM to the extent possible.

5 RELIABILITY-AWARE GARBAGE COLLECTION

Reliability-aware garbage collection places hot and low-risk objects in HBM, and the rest in DRAM. We first provide a general overview of RiskRelief after which we describe the different components in more detail.

5.1 Overview

Figure 3 shows the workflow of RiskRelief. We first profile the Java application to collect per-object read- and write-intensity traces. We then group objects in traces by their allocation site. We use per-object hotness and risk to classify allocation sites as *HBM* to *DRAM*, based on heuristics. This classification constitutes advice which we use to annotate Java bytecodes as *HBM*. All other allocation sites default to *DRAM*. During production, RiskRelief uses a unique allocation sequence for HBM-marked allocation sites. This sequence places hot and low-risk objects in HBM.

Object	Reads	Writes	Method:Idx
O1	16	8	A():10
O2	16	4	A():10
O3	16	0	A():10
O4	4	8	B():14
O5	4	4	B():14
O6	1	0	B():14

(a) Example access trace

Hotness	AVF-X
24	4
20	1
16	0
12	16
8	4
1	0

(b) Hotness and risk calculation

Heuristic	θ_h	θ_t	θ_{hot}	$\theta_{avf\text{-}x}$	HBM Sites
FMID	1%	---	14	4	A
MRAT	1%	20%	24	16	None
MRAT	1%	40%	20	4	A
MRAT	1%	60%	16	4	A
MRAT	1%	80%	12	1	A & B
MRAT	1%	100%	1	0	A & B

(c) Allocation site prediction

Fig. 4. Example of an access trace with allocation sites in the last column (a), per object hotness and AVF-X (b), and prediction of allocation sites using the FMID and MRAT heuristics (c).

5.2 Profiling

RiskRelief relies on offline profiling of Java programs to discover hot and low-risk objects. The outcome of profiling is an *access* trace of per-object reads and writes, see Figure 4 for an example (we will discuss the example in more detail later). We track reads and writes in an architecture-independent manner, i.e., we count all load/store accesses to an object's fields. We count accesses to an object's primitive and reference fields, and to its meta-data header, which contains information such as the class type information, synchronization bits, and garbage collector bits.

Profiling per-object accesses can be done in two ways: (1) using read and write barriers in the managed runtime, or (2) using dynamic instrumentation. All generational garbage collectors use reference write barriers for correctness. Write barriers record all mature-to-nursery pointers in a remembered set, which are processed during a minor collection to precisely identify all live nursery survivors. Primitive write barriers are a straightforward extension of reference write barriers. Unlike write barriers, read barriers incur prohibitive overheads [46]. Most production JVMs include collectors that do not require read barriers. Jikes RVM provides both primitive and reference write barriers [4], but does not implement read barriers.

We therefore rely on dynamic binary instrumentation instead using Pin [45]. Because Pin has no notion of an object's boundary in memory, we deploy a cooperative scheme in which Jikes RVM records each object's starting address, its size in bytes, and its allocation site identifier; in turn, Pin records the number of read and write accesses to each memory location. At the end of the program execution, we gather logs from Jikes RVM and Pin, and we aggregate the two logs to create the access trace which contains all the objects instantiated by each allocation site and the total number of accesses to each object on a per allocation-site basis.

To give each object a unique address in the access trace, we size the mature heap during profiling to preclude full-heap collections. We further set the nursery size to 4 MB. Using this nursery size is a good balance between the size of the access trace and the coverage of mature object behaviors. We label allocation sites with unique identifiers, as in [31].

5.3 Allocation Site Classification

After profiling, we analyze the access trace to generate allocation advice, classifying allocation sites as *HBM* versus *DRAM*. Figure 4(a) shows an example access trace. Two allocation sites contained in methods A() and B() allocate a total of six objects. The trace also shows the different number of reads and writes to objects. We analyze the trace to compute per-object hotness and risk using the definitions described in the previous sections, see Figure 4(b). Next, we use two criteria to label allocation sites: (1) the fraction of total objects allocated from a site that are hot and low-risk, and

(2) heuristics to decide which objects are hot and low-risk. If the fraction of hot and low-risk objects allocated from a site is larger than the homogeneity threshold (θ_h), the site is labeled as *HBM*; otherwise, the site is a *DRAM* site. Next, we use two heuristics to qualify objects as hot versus cold, and low- versus high-risk.

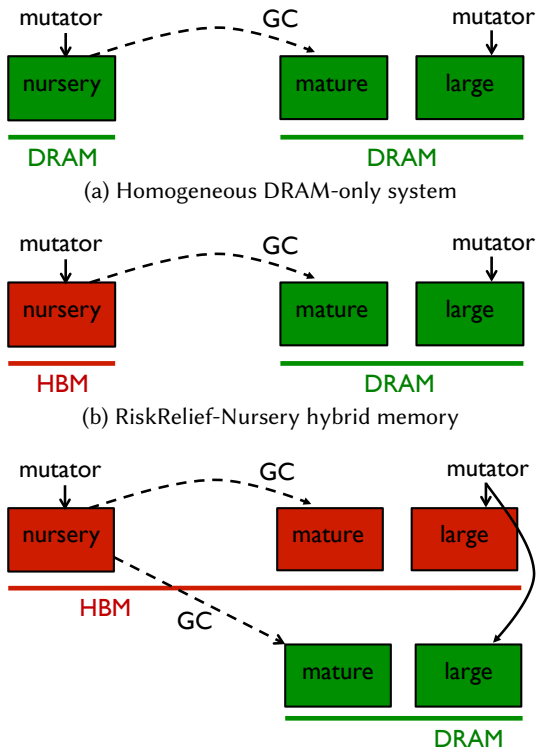
Fixed-Midpoint (FMID) is inspired by Gupta et al. [27] and uses the average hotness (or AVF-X) across all mature space objects as the cut-off to quantify the hotness (or risk) of objects from an allocation site. Specifically, with FMID, we qualify an object as hot if the sum of reads and writes to that object are above the cut-off (average). FMID has the advantage that hotness and AVF-X are straightforward to compute. The disadvantage is that it uses a single cut-off value, which leads to a specific design point in terms of SER, performance and HBM usage. In practice, a heuristic that exposes a trade-off is more desirable, which we advocate in this paper.

Moving-Ratio (MRAT) uses a ratio namely θ_t (e.g., top-10%) to divide objects into two quadrants, e.g., hot and cold. The hotness cut-off (θ_{hot}) places an object allocated from a site in the top-10% of hot objects. Similarly for identifying low-risk objects, the risk cut-off (θ_{avf-x}) places an object within the top-10% low-risk objects. The user or system administrator specifies the ratio based on environmental constraints. Varying the ratio opens up a trade-off between HBM capacity, performance, and overall SER.

Example. Figure 4(a) shows an example access trace consisting of 6 objects from two allocation sites in methods A() and B(), respectively. Per-object hotness and risk is shown in Figure 4(b). We analyze the trace using the FMID and MRAT heuristics, and identify which of the two sites are classified as *HBM* in Figure 4(c). We fix θ_h at 1%, and vary θ_t from 20% to 100% for MRAT. The average hotness and risk is 14 and 4, respectively. Therefore, with FMID, the allocation site in method A() has one object (O1) with hotness larger than the average value, and risk larger than or equal to the average risk. Since 1 out of 3 objects from this site are hot and low-risk, which is higher than the homogeneity-threshold of 1%, this site is classified as HBM. Next, we set θ_t to 20% for MRAT and compute the HBM sites. Since θ_t is 20%, we only consider the hottest object (1 out of 6), and the lowest risk object to compute θ_{hot} and θ_{avf-x}. O1 is the hottest leading to θ_{hot} of 24. Similarly, O4 has the lowest risk, leading to a θ_{avf-x} of 16. Neither allocation site in Figure 4(a) has an object with both hotness larger than or equal to 24, and risk larger than or equal to 16. Thus, using MRAT with θ_t at 20% leads to all allocations in DRAM. On the other hand, setting θ_t to 40% or 60% results in allocation in HBM for A(). Finally, setting θ_t to 80% and 100% results in all allocations in HBM. This example demonstrates the flexibility exposed by MRAT in exploiting the rich trade-offs that exist between SER, performance, and HBM capacity.

5.4 Bytecode Generation

The previous step generates allocation site advice as a file of <site-string, advice> pairs. The advice file only includes the *HBM*-labeled allocation sites. Unlabeled allocation sites default to *DRAM*. Since a minority of allocation sites are labeled *HBM*, the size of the advice file is minimized. We use bytecode rewriting to communicate allocation site labels to the managed runtime. The bytecode rewriter first identifies the allocation site and then queries the advice file to check whether the site is present. If it is not, the rewriter leaves the new bytecode unchanged. If it is, the rewriter overwrites the new bytecode with a newly introduced new_hbm bytecode. The runtime, when interpreting or compiling the new bytecode, uses the default allocator, called ALLOC_DEFAULT. The runtime then copies all objects allocated by such sites to DRAM if they survive a nursery collection. For the new_hbm bytecode, the runtime uses the newly added ALLOC_HBM allocator.



(a) Homogeneous DRAM-only system

(b) RiskRelief-Nursery hybrid memory

(c) RiskRelief-Mature hybrid memory

Fig. 5. Main memory heap organizations.

This allocator sets a bit in the object header which notifies the garbage collector to copy these objects to HBM if they survive a nursery collection.

Note that because RiskRelief is a profile-based approach, there might exist allocation sites that were not seen during profiling, i.e., an allocation site was not executed in the profile run while it gets executed in a production run. These unprofiled sites will be unlabeled, and default to *DRAM*, following the above procedure. Future work may explore whether labeling unprofiled sites as *HBM* might be desirable, or whether dynamically profiling just these objects might be tractable and beneficial.

5.5 Heap Organization

We now describe RiskRelief's heap organizations. The heap organization for a conventional homogeneous DRAM-Only system is shown in Figure 5(a). The RiskRelief collectors place the nursery in HBM because the nursery is highly mutated, and hence contains objects that are both hot and low-risk. RR-N places only the nursery in HBM and the rest, i.e., the mature space and large object space, in DRAM, see Figure 5(b). RR-M further partitions the mature and large object spaces into DRAM and HBM regions, see Figure 5(c).

RR-N operates as follows. Nursery objects are allocated in the HBM nursery. Objects that survive a nursery collection are copied to the mature space in DRAM. Large objects (larger than 8 KB as in

our baseline configuration) are allocated directly in the Large Object Space (LOS) which is mapped in DRAM.

RR-M is more complicated as it requires adjusting the allocation process. In general, new allocation is a two-step process: (1) reserving space and (2) initializing the object header, called post-allocation. For RR-M, post-allocation sets a bit in the object's header if its allocation site is labeled *HBM*, as shown in Figure 6. We steal a bit, not in use from the object header in Jikes RVM, and call it the HBM_BIT. Objects with the HBM_BIT set are predicted to be hot and low-risk. During nursery collection, the garbage collector checks the HBM_BIT of each object. If the bit is set, it promotes the object to the mature space in HBM. Otherwise, it promotes the object to the DRAM mature space.

```
1 @Inline
2 public Address postAlloc(ObjectReference ref, int allocator) {
3   if (allocator == Gen.ALLOC_HBM) {
4     byte old = readHeaderByte(ref);
5     writeHeaderByte(ref, (byte) (old | HBM_BIT));
6   }
7 }
```

Figure 6. Our post-allocation sequence sets the HBM_BIT in the header of (predicted) hot and low-risk objects.

RR-M also involves changes to how large objects are treated. For these objects, RR-M's ALLOC_DEFAULT allocates the object directly in the LOS DRAM space, whereas ALLOC_HBM places the object directly in the LOS HBM space.

6 EXPERIMENTAL SETUP

The main results presented in Section 7 are obtained through detailed architectural simulation to accurately assess performance and reliability. This section elaborates on this methodology. We complement these simulation results with emulation results on real hardware in Section 8.

Java Virtual Machine and workloads. We use Jikes RVM 3.1.2 [5, 6] and nine applications from the DaCapo suite [13] that work with our simulation and VM infrastructure. We use four benchmarks from the DaCapo-9.12-bach benchmark suite (sunflow, lusearch, pmd, and xalan). We use an updated version of lusearch, called lu.Fix [70], that eliminates useless allocation, and an updated version of pmd, called pmd.S [23], that eliminates a scaling bottleneck due to a large input file. We use three benchmarks from DaCapo 2006: fop, antlr and bloat. As in established methodology, we use 2× the minimum heap size for our benchmarks, and we use different inputs for profiling (default) versus measurement (large). We consider 32-instance workloads of our benchmarks to generate realistic memory traffic.

Java performance evaluation. We follow best practices in Java performance evaluation [15, 29, 32]. We use replay compilation to eliminate non-determinism introduced by just-in-time compilation. During a profiling run, the VM records a plan with the optimization level for each method for the run with the shortest execution time. We then run each benchmark for two iterations. In the first unmeasured iteration, the JIT compiler applies the optimization plan to each method. We then measure the second iteration, which excludes compilation overhead and which represents application steady-state behavior. We report the average across four simulation runs.

Simulator. We use Sniper [16] v6.0, a parallel and high-speed cycle-level x86 simulator for multicore systems, using its most detailed cycle-level hardware-validated core model. Prior work extended

Processors	**Parameters**
Number of cores	1 socket, 32 cores
Core frequency	4.0 GHz
Issue width	4-wide out-of-order
ROB size	128 entries
Branch predictor	hybrid local/global predictor
Caches	**Parameters**
L1-I	32 KB, 4 way, 4 cycle access time
L1-D	32 KB, 8 way, 4 cycle access time
L2 cache	256 KB per core, 8 way, 8 cycle
L3 cache	shared 32 MB, 16 way, 30 cycle
HBM	**Parameters**
Capacity	2 GB for hybrid, 32 GB for HBM-only
Bus frequency	500 MHz (DDR 1.0 GHz)
Bus width	128 bits
Channels	8 channels
Banks	8 banks/channel
ECC	SEC-DED ECC [30]
tCAS-tRCD-tRP-tRAS	45-45-45-180 CPU cycles
DRAM	**Parameters**
Capacity	32 GB
Bus frequency	800 MHz (DDR 1.6 GHz)
Bus width	64 bits
Channels	2 channels
Banks	8 banks/channel
ECC	single-ChipKill ECC [21]
tCAS-tRCD-tRP-tRAS	45-45-45-180 CPU cycles

Table 1. Simulated system parameters.

Sniper for managed language runtimes, including dynamic compilation, and emulation of frequently-used system calls [58].

Simulated architectures. We consider a 32-core processor with three memory systems: DRAM-Only, HBM-Only (both with 32 GB of main memory) and a hybrid HBM-DRAM system with 2 GB HBM and 32 GB DRAM, see also Table 1. We emphasize that the 32 GB HBM-Only system is an idealized but unrealistic point of comparison. We further assume a shared 32 MB L3 cache, 25.6 GB/s DRAM bandwidth and 128 GB/s HBM bandwidth. We assume SEC-DED ECC for HBM because of its lower complexity and power consumption [20, 52]. In line with production systems, we assume single-Chipkill ECC for DRAM.

Simulating multi-programmed Java workloads is time-consuming. Specifically, simulating a 32-core system executing 32 instances of the same Java benchmark in rate mode takes up to one month of simulation time for several benchmarks. Moreover, we ran into simulator infrastructure issues when simulating that many cores. We therefore report results for a single-core system with all shared hardware structures scaled down proportionally. We simulate an L3 cache of 1 MB/core, DRAM bandwidth of 0.8 GB/s for each core, and HBM bandwidth of 4 GB/s per core. Our analysis (not shown due to space constraints) confirms that the reported experimental results are conservative — in reality, the improvements in performance and SER through RiskRelief are higher — we confirmed this for up to 8 cores for two benchmarks (pmd and pmd.S) and up to 16 cores for the remaining benchmarks.

Page migration overhead. We compare RiskRelief to the state-of-art reliability-aware OS approach for hybrid memories proposed by Gupta et al. [27]. Page migration overhead is critical to

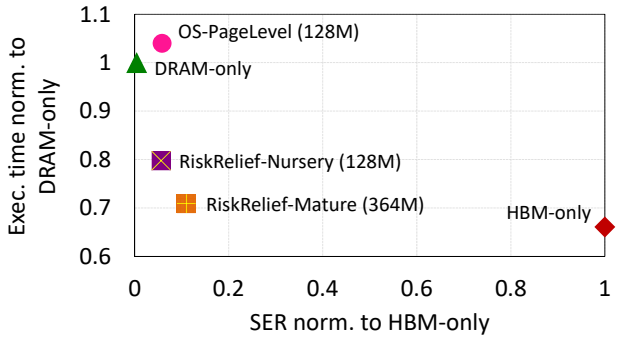


Fig. 7. The execution time versus SER trade-off for the RiskRelief collectors and the state-of-the-art OS approach, normalized to the DRAM-Only and HBM-Only systems.

such OS approaches and includes (1) the latency for moving pages between HBM and DRAM, and vice versa, and (2) TLB shootdown overhead.[1] We assume the latency of copying pages across DRAM and HBM to be 5,000 CPU cycles [9, 24]. The total overhead of a TLB shootdown is independent of the application and depends on the number of cores in the system. The OS keeps track of the 'slave' cores that requested a modified virtual to physical page mapping in the past. During a TLB shootdown, the 'initiator' core requests all slave cores to invalidate the modified TLB entries, flushes its own TLB and waits for the responses from all the slave cores. Following prior work by Villavieja et al. [67], we model the overhead of a TLB shootdown in a system with N cores as follows:

$$T_{shootdown} = N \times T_{slave} + T_{initiator},$$

with T_{slave} and $T_{initiator}$ the time overheads incurred by each slave and initiator cores, respectively. We use published overhead numbers [24] scaled to our 4 GHz processor.

SER calculation. SER, as mentioned before, is computed as the device's raw FIT-Rate times its AVF. We use the default configuration of FaultSim for evaluating our hybrid HBM-DRAM architecture [49]. FaultSim's default transient FIT rate values for DRAM and HBM are based on a field study conducted on the Oak Ridge 'Jaguar' supercomputer [64]. We further assume SEC-DED and single-Chipkill ECC for HBM and DRAM, respectively. Using this methodology, we find that the FIT-Rates equal 0.1140 and 0.0005 for HBM and DRAM, respectively. Our simulation platform precisely computes AVF by counting the number of reads and writes per cache line, which is not possible on real hardware. More specifically, we logically divide memory into 64-byte cache lines and measure the number of reads and writes per cache line, which we then use to compute AVF per cache line. For a hybrid HBM-DRAM system, we first compute the SER for DRAM and HBM as the product of their respective FIT-Rate and AVF. We then scale the individual SER numbers by the percentage of program heap that is placed in DRAM and HBM.

7 RESULTS

We now evaluate RiskRelief collectors across three primary metrics: (1) SER, (2) performance and (3) HBM capacity. Unless otherwise stated, we set θ_h to 1% and θ_t to 20%.

[1]Gupta et al. [27] account for the page migration overhead but not the TLB shootdown overhead.

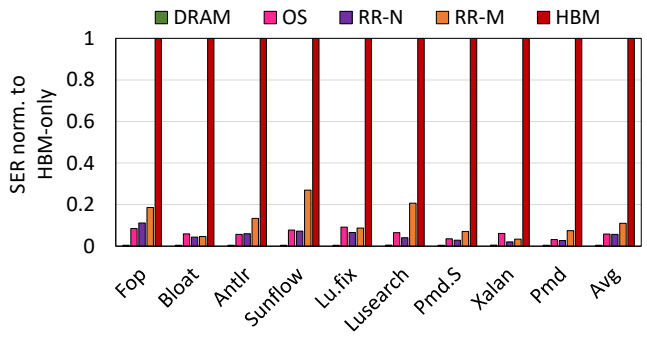


Fig. 8. Soft error rates normalized to HBM-Only for the RiskRelief collectors, the OS approach and DRAM-Only.

7.1 Key Trade-Offs

Using HBM to store a portion of the program heap provides a reliability/performance trade-off, see Figure 7. An HBM-Only system delivers the best performance, but the heap is highly susceptible to soft errors, i.e., the overall normalized SER equals 1. On the other hand, a DRAM-Only system is 34% slower than HBM-Only, but SER is close to 0 (0.003 to be precise). RiskRelief-Nursery places the nursery in HBM and achieves 20% higher performance than a DRAM-Only system. It also reduces SER by 18× compared to an HBM-Only system. HBM capacity for the 32-core system equals 128 MB, which is moderate relative to the total 2 GB HBM capacity.

The state-of-the-art OS approach achieves roughly similar SER as RiskRelief-Nursery, while also requiring 128 MB HBM capacity. Our analysis shows that the OS approach correctly predicts that the nursery is hot and low risk. It thus migrates the nursery pages to HBM. Unfortunately, on x86 multi-core platforms, page migrations incur a substantial performance penalty. The large number of page migrations results in high overhead, and the OS approach performs 24% worse than RiskRelief-Nursery. The significant performance penalty of the OS approach makes it unsuitable for Java applications because the benefits of high HBM bandwidth to access highly mutated and frequently read data is offset by the high cost of page migrations. Our analysis further shows that the overhead of TLB shootdowns is the major contributor to the high cost of page migrations.

Both the state-of-the-art OS approach and RiskRelief-Nursery place the nursery in HBM, using only a modest fraction of the available HBM capacity. Figure 7 shows that the RiskRelief-Mature collector uses a larger fraction of the available HBM capacity by placing part of the mature heap space in HBM as well. RiskRelief-Mature is highly effective at improving performance beyond RiskRelief-Nursery. On average, the execution time reduces by an additional 9% compared to RiskRelief-Nursery, and by 29% compared to a DRAM-Only system, while still improving SER by a factor 9× compared to an HBM-Only system.

7.2 Soft Error Rate

We now discuss soft error rates for the different systems we evaluate in this work. Figure 8 shows SER of DRAM-Only, RR-N, RR-M, and the OS approach, normalized to HBM-Only for the individual workloads. We observe that a DRAM-Only memory system is highly reliable with negligible SER compared to HBM-Only. This observation is consistent with prior work which reports that in DRAM-Only systems, non-DRAM failures, such as those in memory controllers and memory channels, dominate the majority of errors [47]. Whereas HBM-Only is highly unreliable with a

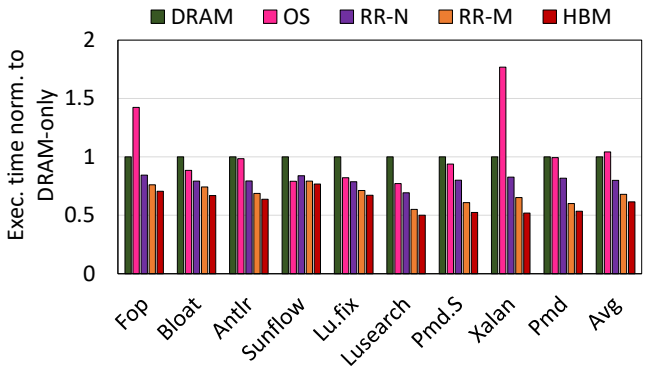


Fig. 9. Execution times normalized to DRAM-Only for the RiskRelief collectors, the OS approach and HBM-Only.

normalized SER of 1, RR-N reduces the SER by 18× on average. All benchmarks observe a reduction in SER and the reduction in SER varies from 9× (Fop) to 48× (Xalan). The differences in per-benchmark SER reduction are due to access patterns in the nursery, more specifically, the ratio of nursery writes to reads. RR-N is the most reliable system of all systems we evaluate in this work, but it does not fully utilize the available HBM capacity. We can utilize the available HBM capacity to gain more performance. As mentioned before, RR-M is the best performing system, however, it sacrifices reliability over RR-N. Still, RR-M reduces SER by 9× over HBM-Only. Some benchmarks, such as Bloat, experience no change in SER reduction with RR-M compared to RR-N. This phenomenon occurs because SER depends on several factors including the ratio of object writes to reads, the rate of memory allocation, and how often the objects in the program heap are accessed after the first allocation. Per-benchmark SER reduction with RR-M compared to HBM-Only varies from 5× to 30×. For completeness, the OS approach achieves a normalized SER that is comparable to RR-N.

7.3 Performance

We show per-benchmark performance results in Figure 9, normalized to a DRAM-Only system. Execution time with an HBM-Only system reduces by 34% on average. Individual benchmarks show a variety of trends. For example, the execution time of Xalan, Pmd, Pmd.S and Lusearch reduces by more than 40%. As reported in Table 3, these benchmarks are characterized by either large heaps, high allocation rates, or high nursery survival rates. The compute-bound Sunflow benefits the least from HBM bandwidth. Our analysis indicates that memory read operations in Sunflow exhibit very high on-chip cache hit rates, thus leading to limited traffic to main memory.

The RiskRelief collectors deliver performance in-between DRAM-Only and HBM-Only. Placing the nursery in HBM with RiskRelief-Nursery (RR-N) reduces execution time by 20% on average compared to a DRAM-Only system. Benchmarks that allocate rapidly benefit more from HBM bandwidth. For example, Lusearch allocates the largest volume of objects across our benchmarks, and RR-N reduces its execution time by 34%. The reasons for this large reduction in execution time include: (1) faster read and write operations to memory, (2) higher throughput of memory zeroing to provide security as guaranteed by Java semantics [1, 70], and (3) faster nursery collections. Surprisingly, Sunflow allocates young objects rapidly in the nursery and has the largest number of nursery collections of all of our benchmarks, yet its execution time reduction with RR-N is

	Migrated Pages			Migration	Migrated Pages
	DRAM→HBM	HBM→DRAM	Total	Epochs	/Epoch
Fop	1037	5	1042	3	347.3
Bloat	1921	494	2415	33	73.2
Antlr	1038	12	1050	7	150.0
Sunflow	1574	556	2130	66	32.3
Lu.fix	1323	23	1346	26	51.8
Lusearch	3584	1522	5106	76	67.2
Pmd.S	1083	42	1125	10	112.5
Xalan	23011	3406	26417	57	463.5
Pmd	2263	111	2374	18	131.9
Avg	4093	686	4779	33	158.8

Table 2. The number of page migrations (DRAM to HBM, HBM to DRAM, and total), the number of 100 ms migration epochs, and the number of page migrations per epoch for the OS approach.

only 15%. This small reduction is because Sunflow has a small nursery survival rate (only 2%) and copying nursery survivors to the mature space does not require high bandwidth. RiskRelief-Mature (RR-M) reduces the execution time on average by an additional 9% over RR-N, and by 29% over a DRAM-Only system. RR-M splits the mature and large object spaces across DRAM and HBM. The benchmark that benefits the most from RR-M is Lusearch. The execution time of Lusearch reduces by 43%. The performance of Lusearch with RR-M is only 5% less compared to HBM-Only, showing the effectiveness of RR-M in exploiting HBM's high bandwidth. Similarly, the performance of Xalan and the two variants of Pmd also improve substantially with RR-M.

The OS approach leads to a significant performance degradation compared to RR-N. Performance degrades for most benchmarks and we note a significant performance degradation for Fop (42%) and Xalan (77%). The reason is the high number of page migrations per unit of time, see also Table 2 which reports the number of page migrations from DRAM to HBM and vice versa, the number of 100 ms migration epochs, and the number of page migrations per epoch. We note that Fop and Xalan are the benchmarks with the highest number of page migrations per unit of time: 347.3 and 463.5 migrations per epoch. Each page migration incurs the overhead of copying the pages and TLB shootdowns. Our measurements indicate that TLB shootdowns account for 41% and 45% of the total execution time for Fop and Xalan, respectively. In other words, TLB shootdowns lead to significant performance degradations for workloads that incur a large number of page migrations per unit of time. The OS approach delivers performance that is better than RR-N for Sunflow, and only slightly worse than RR-N for Bloat, Lu.fix and Lusearch. This is due to the relatively small number of page migrations per 100 ms epoch for these benchmarks, see Table 2. The number of page migrations per epoch is substantially smaller for these benchmarks — Sunflow (32.3), Bloat (73.2), Lu.fix (51.8) and Lusearch (67.2) — compared to the other benchmarks with more than one hundred and up to several hundreds of page migrations per epoch; note that Sunflow has the lowest number of page migrations which leads to a small performance overhead (6%) and a net performance improvement over RR-N.

7.4 RR-M versus Performance-Focused GC

Utilizing HBM capacity is a trade-off between performance and reliability. RR-M can be configured in a variety of ways to exploit this trade-off space. Performance improves when RR-M is configured to place more mature-space objects in HBM, but this compromises reliability. We show this trade-off in Figure 10. We vary θ_t from 10% to 40%. Execution time reduces by 3%, but the SER increases by 5.4×. The reason for the SER increase is that, as RR-M tries to achieve higher performance by

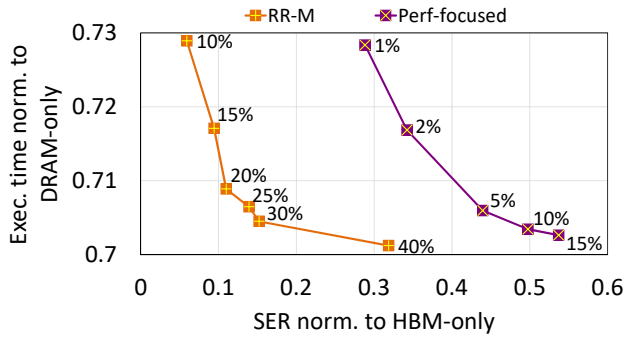


Fig. 10. Execution time versus SER trade-off for different configurations of RR-M and its performance-focused variant.

	Allocation	Heap	RR nursery	RR-M-10%		RR-M-20%		RR-M-30%		RR-M-40%		OS-PageLevel	
	MB	MB	survival %	avg	max	avg	max	avg	max	avg	max	avg	max
Fop	1792	2560	20%	169	184	215	269	230	296	235	302	106	129
Bloat	39872	2112	4%	162	180	167	190	167	187	185	214	128	168
Antlr	7872	1536	15%	250	341	276	393	324	491	339	518	128	128
Sunflow	61440	3456	2%	190	245	410	707	479	850	484	860	124	127
Lu.fix	27136	2176	2%	177	217	177	215	177	216	191	244	126	156
Lusearch	137408	2176	4%	972	1478	959	1474	966	1487	960	1491	121	156
Pmd.S	6464	3136	27%	177	215	327	520	389	659	418	723	128	129
Xalan	31360	3456	14%	210	278	301	432	320	437	322	435	168	515
Pmd	11648	3136	23%	335	513	449	730	608	1019	677	1150	122	140
Avg	36110	2638	12%	294	406	364	548	407	627	423	660	128	183
Heap %				12%		18%		21%		23%		5%	

Table 3. Object demographics: total allocation, heap size, nursery survival rates, and average and maximum mature heap usage (in MB) for our 32-instance workloads.

placing an increasingly larger fraction of the mature space in HBM, it copies objects with low AVF to HBM. In other words, as θ_t increases, allocation sites with a larger number of high-risk objects are classified as HBM, which results in higher performance, but lower reliability.

Figure 10 plots a similar performance versus reliability trade-off curve for a performance-focused variant of RR-M. This performance-focused variant labels allocation sites as HBM based only on the percentage of hot objects allocated from the site. Similar to RR-M, it uses the θ_t threshold to classify objects as hot versus cold. The resulting trade-off curve with this performance-focused collector clearly shows the benefits of RiskRelief in mitigating HBM's high susceptibility to soft errors. Specifically, for the same performance, RR-M exhibits 4.8× lower SER than the performance-focused variant. RR-M takes into account both how often an object is accessed and its AVF before placing it in HBM.

7.5 Memory and Demographic Analysis

Table 3 summarizes total allocation, nursery survival rates, and percentage of mature heap in HBM for RR-M for the different 32-instance workloads. Our applications allocate frequently ranging from 1.8 GB (Fop) up to 137 GB (Lusearch). Our nursery survival rates vary from 2% to 27%. Copying objects to HBM is faster than DRAM, and hence benchmarks that copy a larger fraction of objects to HBM on a nursery collection benefit more from HBM's high bandwidth. Examples include

Xalan and Pmd. The next columns show the average and maximum HBM (in MB) for different configurations of RR-M. Specifically, we show the HBM capacity in MB for different θ_t thresholds. HBM capacity with the most reliable RR-M configuration (θ_t of 10%) equals 294 MB on average, and up to 972 MB. Lusearch consumes the largest HBM capacity with close to 1.5 GB. RR-M places only 12% of the total heap volume in HBM with a 10% θ_t threshold. The percentage of heap volume in HBM increases to 23% of the total heap volume in HBM with a θ_t of 40%. On the other hand, the OS approach places only 5% of the total heap in HBM.

8 EVALUATION ON REAL HARDWARE

Accurately assessing SER for a hybrid memory systems requires per-cacheline read/write statistics which we can only obtain through simulation. We now complement our simulation results with experimentation on commercial hardware, for three reasons: (1) to demonstrate that we can deploy RiskRelief on real systems, (2) to show that RiskRelief directs the vast majority of writes to HBM, and (3) to report the runtime overhead of RR-M relative to RR-N.

Emulation platform. Since we lack access to a commercial machine with HBM, we emulate hybrid HBM-DRAM memory on an existing multi-socket NUMA platform, as in [3]. Commercial HBM systems present HBM as an additional NUMA node to the OS [53], which is exactly what we emulate. In other words, by running Java workloads on the emulation platform with RiskRelief collectors, we incorporate OS and runtime effects as expected on commercial HBM systems. We isolate the Java workload on one socket and disable the other socket. We populate both sockets with commodity DRAM chips. Local memory emulates HBM, and remote memory emulates DRAM. We modify Jikes' MMTk to split the virtual heap into HBM and DRAM. Our two-socket Intel Sandy Bridge E5-2650L processor has 8 physical cores per socket and two hyperthreads per core. We use Ubuntu 12.04.2 with a 3.16.0 kernel. We run 8-instance workloads to utilize all the available cores.

Number of writes to HBM. We now quantify the number of writes to HBM versus DRAM on the emulation platform which features 132 GB of main memory, evenly distributed between the two sockets. We use all DRAM channels on both sockets. All cores share the 20 MB LLC on each processor. The available bandwidth to memory is 51.2 GB/s, more than the maximum bandwidth consumed by any of our workloads. A QPI link that supports up to 8 GB/s connects the two sockets. We use Intel's pcm-memory utility to measure the number of writes to HBM and DRAM.

RiskRelief allocates the frequently accessed low-risk objects in HBM and the rest in DRAM. We thus expect that most writes happen to HBM. We observe that in simulation, on average, 90% and 87% of writes happen to HBM for RR-M and RR-N, respectively. On the emulation platform, we find that 87% and 83% of writes happen to HBM, respectively. Simulation and emulation thus confirm that RiskRelief captures the vast majority of writes to HBM — this indicates that the frequently accessed low-risk objects are indeed allocated in HBM. The small discrepancy between emulation and simulation is a result of differences in the OS, hardware prefetcher, memory controller, among other things.

RR-M runtime overhead. RR-M incurs runtime overhead because of the extra steps involved during post-allocation and nursery evacuation. To quantify these overheads as accurately as possible, we compare the performance of RR-M versus RR-N on the emulation platform while placing the entire heap on one socket of our NUMA platform. On average, the overhead incurred by RR-M is less than 1%, with a maximum of 1.3% for lusearch.

9 OTHER RELATED WORK

Beyond the related work already discussed in this paper, some prior work focuses on automated memory management for hybrid DRAM-PCM memories. However, to the best of our knowledge,

this is the first work to automatically manage memory to improve soft error reliability in 3D-stacked memories by dynamically allocating objects to HBM versus DRAM through garbage collection in the managed language runtime.

Production systems now combine DRAM with non-volatile memory (NVM) to deliver high capacity and performance. The most promising NVM, Phase Change Memory (PCM), suffers from low write endurance. Gao et al. [26] use hardware and OS cooperation to expose defective lines in PCM to the garbage collector to avoid allocation in defective lines.

Write-rationing garbage collection for hybrid DRAM-PCM memories [4] places frequently written objects in DRAM to protect PCM from writes and extend its lifetime. More specifically, Kingsguard-Nursery places the nursery in DRAM because the nursery is highly mutated. Kingsguard-Writers dynamically monitors objects to discover highly written mature objects. Crystal Gazer exploits offline profiling to identify allocation sites that produce highly written objects [2].

Wang et al. [68] focus on Big Data systems (e.g., Spark) and leverage GC to place highly accessed information in DRAM in hybrid DRAM-PCM systems. They exploit memory semantics in the Java runtime and focus solely on performance.

10 CONCLUSION

Emerging high-bandwidth memory (HBM) uses 3D stacking to offer more bandwidth than DRAM. Unfortunately, its capacity is limited, and soft error rate is high. Due to greater bit density and new failure modes, hardware error correction alone is insufficient to make HBM reliable. Prior software approaches that leverage the OS to place hot and low-risk pages in HBM have several drawbacks as they operate at a coarse-grained page granularity and introduce page migration overheads that are prohibitive for multicore systems.

This work explores garbage collection in managed runtimes to balance reliability and performance for a hybrid HBM-DRAM memory system. We propose reliability-aware garbage collection to allocate fine-grained hot and low-risk objects in HBM. Both RiskRelief-Nursery and RiskRelief-Mature place the nursery for young objects in HBM because the nursery is highly accessed and low-risk. RiskRelief-Mature further uses allocation-site prediction to map hot and low-risk mature objects in HBM. We show that object hotness and risk are weakly correlated. RiskRelief-Mature thus uses heuristics to classify objects as hot *and* low-risk for allocation in HBM. Reliability-aware garbage collection substantially outperforms the state-of-the-art OS approach, substantially improves SER over an HBM-Only system, and significantly improves performance over a DRAM-Only system. This work shows that exposing 3D stacking to language runtimes is a promising avenue for balancing reliability and performance.

REFERENCES

[1] Shoaib Akram, Jennifer B. Sartor, and Lieven Eeckhout. 2016. DVFS performance prediction for managed multithreaded applications. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 12–23.

[2] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2019. Crystal Gazer: Profile-Driven Write-Rationing Garbage Collection for Hybrid Memories. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 1 (2019), 1–27.

[3] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2019. Emulating and Evaluating Hybrid Memory for Managed Languages on NUMA Hardware. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 93–105.

[4] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-rationing Garbage Collection for Hybrid Memories. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 62–77.

[5] Bowen Alpern, C. Richard Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton

Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. 2000. The Jalapeño virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238.

[6] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. 2005. The Jikes RVM Project: Building an Open Source Research Community. *IBM System Journal* 44, 2 (2005), 399–418.

[7] AMD. [n.d.]. *High Bandwidth Memory*. AMD. https://www.amd.com/en/technologies/hbm

[8] Andrew W. Appel. 1989. Simple Generational Garbage Collection and Fast Allocation. *Software: Practice and experience* 19, 2 (1989), 171–183.

[9] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. 2017. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 273–287.

[10] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh1, Don McCauley, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. 2006. Die Stacking (3D) Microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 469–479.

[11] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and Realities: The Performance Impact of Garbage Collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 25–36.

[12] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 137–146.

[13] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*. 169–190.

[14] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 22–32.

[15] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2008. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Commun. ACM* 51, 8 (2008), 83–89.

[16] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 3 (2014), 1–25.

[17] ChiaChen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2014. CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12.

[18] ChiaChen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2015. BEAR: Techniques for mitigating bandwidth bloat in gigascale DRAM caches. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. 198–210.

[19] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2017. BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with stacked-DRAM. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. 268–280.

[20] NVIDIA Corp. 2016. *NVIDIA Pascal Architecture*. NVIDIA Corp. https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/

[21] Timothy J. Dell. 1997. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory by.

[22] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. 2010. Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

[23] Kristof Du Bois, Jennifer B. Sartor, Stijn Eyerman, and Lieven Eeckhout. 2013. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 355–372.

[24] Bogdan F. Romanescu, Alvin R. Lebeck, Daniel J. Sorin, and Anne Bracy. 2010. UNified Instruction/Translation/Data (UNITD) coherence: One protocol to rule them all. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*. 1–12.

[25] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. 2009. Demystifying Magic: High-level Low-level Programming. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*. 81–90.

[26] Tiejun Gao, Karin Strauss, Stephen M. Blackburn, Kathryn S. McKinley, Doug Burger, and James Larus. 2013. Using Managed Runtime Systems to Tolerate Holes in Wearable Memories. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 297–308.

[27] Manish Gupta, Vilas Sridharan, David Roberts, Andreas Prodromou, Ashish Venkat, Dean Tullsen, and Rajesh Gupta. 2018. Reliability-Aware Data Placement for Heterogeneous Memory Architecture. In *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 583–595.

[28] Gabriel H. Loh and Mark D. Hill. 2011. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 454–564.

[29] Jungwoo Ha, Magnus Gustafsson, Stephen M. Blackburn, and Kathryn S. McKinley. 2008. Microarchitectural Characterization of Production JVMs and Java Workloads. In *IBM CAS Workshop*.

[30] Mu-Yue Hsiao. 1970. A Class of Optimal Minimum Odd-weight-column SEC-DED Codes. *IBM Journal of Research and Development* 14, 4 (1970), 395–401.

[31] Jipeng Huang and Michael D. Bond. 2013. Efficient Context Sensitivity for Dynamic Analyses via Calling Context Uptrees and Customized Memory Management. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 53–72.

[32] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The Garbage Collection Advantage: Improving Mutator Locality. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 69–80.

[33] ITRS. 2005. Internatial Technology Roadmap for Semiconductors: ASSEMBLY AND PACKAGING.

[34] Prashant J. Nair, David A. Roberts, and Moinuddin K. Qureshi. 2014. Citadel: Efficiently Protecting Stacked Memory from Large Granularity Failures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 51–62.

[35] JEDEC. [n.d.]. *High Bandwidth Memory*. JEDEC. https://www.jedec.org/standards-documents/docs/jesd235a

[36] Hyeran Jeon, Gabriel H. Loh, and Murali Annavaram. 2014. Efficient RAS support for die-stacked DRAM. In *Proceedings of the International Test Conference (ITC)*. 1–10.

[37] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 25–37.

[38] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. 404–415.

[39] Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravishankar Iyer, Srihari Makineni, Donald Newell, Yan Solihin, and Rajeev Balasubramonian. 2010. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*. 1–12.

[40] Richard Jones and Rafael Lins. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc.

[41] Moinuddin K. Qureshi and Gabe H. Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 235–246.

[42] 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 *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. 361–372.

[43] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W. Leet. 2015. A fully associative, tagless DRAM cache. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. 211–222.

[44] Xiao Liu, David Roberts, Rachata Ausavarungnirun, Onur Mutlu, and Jishen Zhao. 2019. Binary Star: Coordinated Reliability in Heterogeneous Memory Systems for High Performance and Scalability. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (MICRO)*. 807–820.

[45] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 190–200.

[46] Matthias Meyer. 2006. A True Hardware Read Barrier. In *Proceedings of the 5th International Symposium on Memory Management (ISMM)*. 3–16.

[47] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. 2015. Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 415–526.

[48] Micron. 2007. TN-41-01: Calculating memory system power for DDR3.

[49] Prashant J. Nair, David A. Roberts, and Moinuddin K. Qureshi. 2015. FaultSim: A Fast, Configurable Memory-Reliability Simulator for Conventional and 3D-Stacked Systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2015), 1–24.

[50] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-performance Big-data-friendly Garbage Collector. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 349–365.

[51] Mark Oskin and Gabriel H. Loh. 2015. A Software-Managed Approach to Die-Stacked DRAM. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*. 188–200.

[52] Mike O'Connor. 2014. Highlights of the high-bandwidth memory (hbm) standard. In *Memory Forum Workshop*.

[53] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis. 2017. Exploring the Performance Benefit of Hybrid Memory System on HPC Environments. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 683–692.

[54] Andreas Prodromou, Mitesh Meswani, Nuwan Jayasena, Gabriel Loh, and Dean M. Tullsen. 2017. MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories. In *Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 433–444.

[55] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA)*. 126–136.

[56] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. 2009. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*. 371–382.

[57] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 29–40.

[58] Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S. McKinley. 2014. Cooperative Cache Scrubbing. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*. 15–26.

[59] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. 2009. DRAM Errors in the Wild: A Large-scale Field Study. *ACM SIGMETRICS Performance Evaluation Review* 37, 1 (2009), 193–204.

[60] Rifat Shahriyar, Stephen M. Blackburn, Xi Yang, and Kathryn S. McKinley. 2013. Taking Off the Gloves with Reference Counting Immix. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 93–110.

[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 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 247–257.

[62] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. 2014. Transparent Hardware Management of Stacked DRAM as Part of Memory. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 13–24.

[63] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 297–310.

[64] Vilas Sridharan and Dean Liberty. 2012. A study of DRAM failures in the field. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

[65] David Ungar. 1984. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE)*. 157–167.

[66] David Ungar and Frank Jackson. 1992. An Adaptive Tenuring Policy for Generation Scavengers. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 14, 1 (1992), 1–27.

[67] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. 2011. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*

(PACT). 340–349.

[68] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 347–362.

[69] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. 2012. Barriers Reconsidered, Friendlier Still!. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*. 37–48.

[70] Xi Yang, Stephen M Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S McKinley. 2011. Why Nothing Matters: The Impact of Zeroing. In *Proceedings of the ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 307–324.

[71] Vinson Young, Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2018. ACCORD: Enabling Associativity for Gigascale DRAM Caches by Coordinating Way-Install and Way-Prediction. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. 328–339.

[72] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. 2009. Allocation Wall: A Limiting Factor of Java Applications on Emerging Multi-core Platforms. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 361–376.