Managing Hybrid Memories by Predicting Object Write Intensity

Shoaib Akram
Ghent University
Belgium

Kathryn S. McKinley
Google
United States of America

Jennifer B. Sartor
Vrije Universiteit Brussel and Ghent University
Belgium

Lieven Eeckhout
Ghent University
Belgium

ABSTRACT

Emerging Non-Volatile Memory (NVM) technologies offer more capacity and energy efficiency than DRAM, but their write endurance is lower and latency is higher. *Hybrid memories* seek the best of both worlds — scalability, efficiency, and performance — by combining DRAM and NVM. Our work proposes modifying a standard managed language runtime to allocate objects either in DRAM or NVM to maximize the use of NVM capacity without wearing it out. The key to our approach is correctly predicting highly mutated objects and allocating them in DRAM and allocating rarely mutated objects in NVM. We explore *write-intensity* prediction based on object (1) size, (2) class type, and (3) allocation site. We find predictions using allocation site are the most accurate.

Our memory manager for hybrid memories consists of (1) an offline profiling phase that produces placement advice on a per allocation-site basis, and (2) a garbage collector that allocates mature objects in DRAM or NVM based on this advice and that allocates highly mutated nursery objects in DRAM. We explore two heuristics for classifying sites as write-intensive (DRAM) or rarely written (NVM). Write-Frequency (FREQ) uses the number of writes to objects allocated by each site. Although it can limit writes to NVM up to 1% and 3%, it allocates just 50% to 20% of mature objects in DRAM. *Write-Density* (DENS) computes number of writes to objects relative to object size. Write-Density is a better predictor. When it limits NVM writes to 2%, it can allocate 88% of mature objects to NVM. Pareto optimal configurations may increase writes to 10% and store 99% of mature objects in NVM. Using write-intensity predictors to proactively place objects in hybrid DRAM and NVM memory systems prolongs NVM's lifetime while exploiting its capacity.

<Programming'18> Companion, April 9–12, 2018, Nice, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5513-1/18/04…$15.00
https://doi.org/10.1145/3191697.3213803

CCS CONCEPTS

• **Information systems** → **Phase change memory**; • **Computer systems organization** → **Architectures**; • **Hardware** → **Non-volatile memory**; • **Software and its engineering** → **Garbage collection**;

KEYWORDS

Managed runtimes, Garbage collection, Non-volatile memory (NVM), Hybrid DRAM-NVM memories, Write-intensity prediction

ACM REFERENCE FORMAT:

Shoaib Akram, Kathryn S. McKinley, Jennifer B. Sartor, and Lieven Eeckhout. 2018. Managing Hybrid Memories by Predicting Object Write Intensity. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3191697.3213803

1 INTRODUCTION

Modern applications demand greater main memory capacity for fulfilling user needs. Unfortunately, DRAM scaling has slowed [17, 18], leading researchers to explore alternative technologies. Non-Volatile Memory (NVM) exhibits promising features: byte addressability, high density, scalability (capacity), negligible idle power, and non-volatility [15]. A promising technology is Phase Change Memory (PCM), but it has shortcomings: high access latency, write latency exceeds read latency, high write energy, and low write endurance [16, 19].

Improvements in manufacturing technology are bridging the latency gap [14]. Finite write endurance is however more challenging because each write changes the material form [9]. To make NVM practical, systems need to exploit NVM's capacity without compromising its lifetime. Prior work proposes *hybrid memory* that combines DRAM and PCM [16, 19] to achieve the best of both approaches: low latency, high capacity, energy efficiency and durability. Prior hardware and OS solutions mitigate writes to NVM by reactively placing highly mutated pages in DRAM [16, 19, 20, 25]. These solutions assume existing programming models, although hybrid memories can support a mix of existing and new programming models, including using PCM as a disk replacement and persistent heap data structures [11, 22]. This paper also focuses on exploiting hybrid memories with existing programming models and thus requires no additional programming effort.

We propose a runtime that automatically manages allocation of objects in DRAM and NVM. We exploit the memory manager

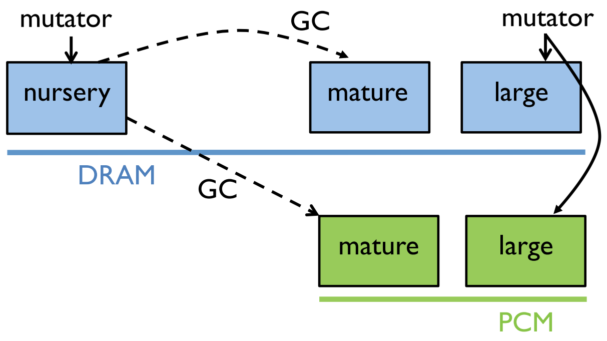


Figure 1: Heap organization for hybrid memories. ***Highly mutated nursery survivors are promoted to DRAM during a garbage collection. The mutator allocates highly mutated large objects in DRAM.***

(garbage collector) in managed runtimes for languages such as Java, C#, JavaScript, and Python. Our goal is to place as much of the program's dynamic heap in NVM as possible to exploit NVM capacity, while minimizing writes to NVM to prevent wear out. Our hypothesis is that most objects are not highly written and thus the goal of this proposed division is possible. An empirical analysis of write behaviors in Java applications motivates our approach. We find that, on average, 70% of the writes in Java applications occur to nursery objects, motivating a design that allocates them to DRAM. Of the writes to mature objects, 2% of mature objects sustain 80% of these writes. In this paper, we propose a profiling mechanism to predict mature object behaviors and a garbage collector that promotes mature objects to DRAM or NVM based on its predictions.

Our approach exposes the physical memory organization to the runtime. Figure 1 shows how our proposed heap organization maps on to hybrid DRAM and PCM memories. We modify the JVM to explicitly request DRAM and NVM from the OS. The mutator allocates new objects in a DRAM nursery, because the nursery is highly mutated. During a garbage collection (GC), nursery survivors are promoted to either DRAM or NVM. Large objects that do not fit in the nursery are allocated directly in the mature space. We use write-intensity prediction to promote mature objects and allocate large objects into DRAM or NVM. The predictor identifies write-intensive objects, which on average for our applications, constitute less than 10% of all mature objects.

We measure writes to objects in 12 Java applications, and find that objects originating from the same allocation site in a program show uniform write behavior. Most allocation sites are dominated by either write-intensive objects or rarely written objects. We classify sites as write-intensive (DRAM) or not (NVM). During production execution, the garbage collector exploits this classification for allocating large objects, and promoting nursery survivors to either DRAM or NVM. We also try other prediction mechanisms, such as object size and class-type, but they are less accurate.

We evaluate two heuristics. Both require allocation-site homogeneity. A site is classified as DRAM if a fraction of objects allocated from it are write intensive. To identify write-intensive objects, Write-Frequency (FREQ) counts the number of writes to individual objects. Objects that get more than a threshold of writes are write intensive. FREQ eliminates 90% of the writes to the NVM mature space by placing 6% of the mature heap in DRAM. *Write-Density* (DENS) computes write-intensity by dividing writes to an object by object size in bytes. Objects with density more than a cutoff density are write intensive. DENS eliminates 90% of the writes to NVM mature space and places 1% of the mature heap in DRAM. By taking into account both object size and writes, DENS thus results in a higher NVM utilization.

In summary, the contributions of this paper are:

- demonstrating that writes to mature objects in Java are predictable on a per allocation-site basis;
- using profiling information to predict sites as write-intensive (DRAM) or not (NVM);
- a garbage collector that uses write-intensity prediction to allocate objects in DRAM and NVM; and
- results that show this approach can limit writes to NVM while still exploiting the capacity advantage of NVM.

This paper leaves as an open question performance of hybrid memories, and questions such as if highly read objects will also need to reside in DRAM to meet application latency requirements. As memory systems evolve to meet application needs, we believe the hardware, operating system, and language runtime implementations will all have a role to play.

2 RELATED WORK

This section discusses prior architecture and OS work on managing hybrid memories and on object prediction.

Hardware and OS techniques for hybrid memories. The vast majority of prior work to eliminate NVM writes divides into two categories: (1) hardware that uses DRAM as a cache for frequently accessed pages in NVM [16, 19], and (2) OS placement of highly-mutated pages in DRAM using page migrations [20, 26]. The OS techniques rank pages according to their write frequency. The OS migrates the top highly written pages to DRAM based on write thresholds. In existing approaches, the write threshold is pre-determined. If a DRAM resident page does not incur writes, the OS moves it back to NVM. Prior hardware and OS techniques are reactive, and work at the page granularity. A significant disadvantage of these prior approaches is page migrations result in additional writes to NVM.

Optimizing performance of hybrid memories. Recent work uses offline profiling of C programs to find allocation sites that produce memory with high access frequency [25]. Initial placement of heap data is guided by allocation-site information gathered offline. During runtime, the OS monitors and moves pages between DRAM and NVM. This approach is limited by C semantics because objects can not move. Furthermore, our approach is more fine-grain — it places individual objects instead of pages.

Recent work uses the managed runtime to optimize for performance in hybrid memories, as opposed to lifetime [24]. It performs an offline profiling phase to identify object allocation sites for cold (rarely read or written) and hot old objects. It places all nursery objects in DRAM. It promotes nursery survivors according to their tag, moving hot objects to DRAM and cold ones to PCM. In contrast, our work optimizes PCM lifetime.

Object behavior prediction. Prior work predicts object behaviors to: (1) colocate objects with similar lifetimes for fast allocation and deallocation, and (2) colocate frequently referenced objects to improve memory locality [2, 6, 8, 10, 21]. Prior work on programs written in C predict object lifetimes and reference behaviors using a combination of calling context and size [2, 21].

For managed languages, generational collectors perform age-based segregation for fast memory allocation and reclamation. Pretenuring allocates long-lived objects directly in the mature space, which improves performance by eliminating copying. Prior work uses allocation site to accurately predict object lifetimes for pretenuring Java objects [6, 8]. Similarly, both this prior work and our approach gather a profile, build a predictor, and use prediction in a separate execution. In contrast, instead of predicting object lifetimes, we show allocation site is a good predictor of the write intensity of Java objects.

3 BACKGROUND

This section presents background on generational garbage collection and object prediction at call sites.

Generational Garbage Collection. Empirical studies show that most objects die young [23]. High-performance collectors today segregate objects based on their age. The mutator allocates new objects in a contiguous nursery. When the nursery is full, a *minor* collection identifies live nursery objects by tracing the roots. Any nursery objects reachable from the roots is live and is copied to a mature space. All nursery memory is reclaimed en masse for fresh allocation. When the mature space is full, a full-heap (mature) collection collects the entire heap.

Nursery size affects overall performance, pause time, and memory footprint. Large nurseries give objects more time to die and sometimes improve performance at the expense of increased pause time and memory footprint. We use a 4 MB nursery which delivers good performance for our applications and collector. The nursery is highly-mutated because freshly allocated objects are zeroed and then initialized.

Write barriers. We use write barriers to understand and measure writes to mature objects. The compiler adds instrumentation code on every write for gathering write statistics. We piggyback on the write barrier in generational collectors that records pointer references from mature objects to nursery objects in order to independently collect the nursery. For gathering write statistics however, we instrument writes to both pointers and primitives.

Large objects. Copying nursery survivors to the mature space is a major performance overhead in generational collectors. Large objects can rapidly exhaust the nursery space leading to frequent minor collections. Most JVMs therefore allocate large objects in a separate non-copying space in the heap. Jikes RVM uses a heuristic to determine the large object threshold. Because large objects are not allocated in the nursery, we consider them in our analysis of writes to objects.

Object behavior prediction. We use profile based object behavior prediction, which measures characteristics such as type, class, lifetime, and in our case writes, with respect to allocation sites and then applies some action in a later execution.

Allocation-time information. Allocation-time information includes the object's size, class (type), allocation site, and calling context. Context is the dynamic sequence of methods leading to a site. Because we find calling-context only slightly improves write-intensity prediction accuracy over allocation site, we do not discuss it further.

4 METHODOLOGY

This section describes our methodology for profiling and predicting write intensity, including the JVM, applications, write barriers, and collector configuration.

Java Virtual Machine. We use Jikes RVM 3.1.2, a Java-in-Java research VM that includes a modular memory management toolkit [1, 3]. The easy-to-modify garbage collectors and write barriers make Jikes RVM an ideal experimentation platform [1, 4, 12]. We configure Jikes RVM with replay compilation to eliminate non-determinism introduced by just-in-time compilation. During the first unmeasured iteration, the JIT compiler applies a pre-recorded optimization plan to each method. We gather write-intensity traces and take measurements during the second stable iteration.

Computing site identifiers. The size and type of an object are available at allocation time. We add an extra word to the object header for storing the allocation site during the profiling run. We use the calling context profiling patch from Huang et al. [13] to compute site identifiers. The compiler computes a unique identifier for each allocation-site, and generates instrumentation to store the identifier in the object header during execution.

Gathering write-intensity traces. To correlate allocation-time information to the write behavior of mature objects, we first produce a write-intensity trace of an application. For ease of implementation and analysis, we use a non-moving mark-sweep mature space and a copying nursery. Each mature object thus has a unique address in the heap.

The JVM uses a counter to record mature-object writes, except for the initializing writes, along with the object size, type, and allocation site. We obtain traces using a 4 MB nursery and an unlimited mature space. During the profiling run, a write barrier first inserts any previously unwritten object, its size, type and allocation site in a hash table. It then increments the object write counter. When the profiling run finishes, the contents of the hash table forms the write-intensity trace. The table is indexed using object addresses.

Java applications. We use 12 Java applications: 11 from DaCapo [5], plus pseudojbb2005 (pjbb) [7]. We use their default datasets. Table 1 lists the Java applications and heap sizes we use. In addition to the original versions of lusearch and pmd in DaCapo, we use an updated version of lusearch, called lu.Fix, that eliminates useless allocation, and an updated version of pmd, called pmd.S that eliminates a scaling bottleneck due to a large input file.

5 WRITE-INTENSITY PREDICTION

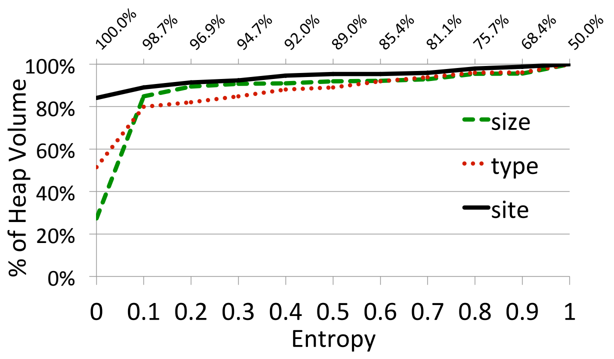
This section presents our write homogeneity metric and prediction mechanisms. We then describe how the collector exploits this information to allocate objects in hybrid memories.

5.1 Write Homogeneity

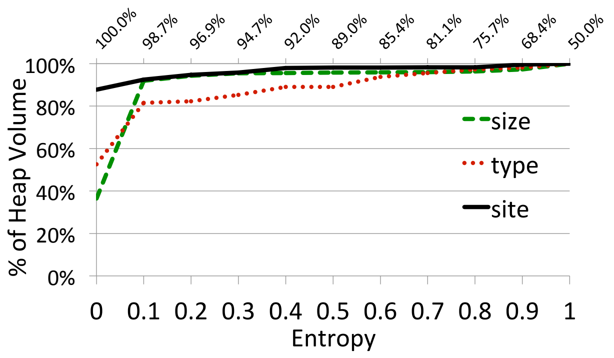
Computing write homogeneity. To discover if allocation-time information will accurately predict the write-intensive objects, we first measure write homogeneity in our applications. We analyze the write-intensity traces (see Section 4) by grouping together objects based on type, size, and site at allocation time. When analyzing object type, size and site, we first identify all the unique types, sizes, and allocations (metrics). We then measure the distribution of write-intensive versus other objects for each metric. Using these

Table 1: The heap sizes we use, and the number of unique sizes, types and sites in our benchmarks.

	Fop	Luindex	Antlr	Bloat	Jython	Xalan	Pmd	Pmd.S	Lusearch	Lu.Fix	Sunflow	Pjbb	Average
Heap (MB)	80	44	48	66	80	108	98	98	68	68	108	400	106
# Sizes	110	74	245	190	181	75	242	231	67	66	55	157	109
# Types	246	207	147	253	298	220	436	423	166	170	144	135	240
# Sites	427	341	374	714	741	391	670	640	247	250	240	286	444



(a) Write-Frequency Threshold = 1K



(b) Cutoff Density = 1

Figure 2: Homogeneity of object writes on a per size, class-type, and allocation-site basis. ***The homogeneity of allocation-site is high: 90% of the heap volume originates from sites that have 90% of the same object kind.***

distributions, we compute the entropy of each metric. The entropy of a type is defined as as follows:

$$E = -(O_w \times \log_2 O_w) - (O_{nw} \times \log_2 O_{nw}) \quad (1)$$

O_w is the fraction of write-intensive objects and O_{nw} is the fraction of other objects. Similar to types, we compute the entropy of all sizes and allocation sites. We use two criteria to identify write-intensive objects: (1) *write-frequency:* the object gets more than a threshold of number of writes, and (2) *write-density:* the object has more than a threshold writes per byte.

Results. Figure 2 shows homogeneity curves averaged over all benchmarks. We use a write-intensity threshold (w_t) of 1K in Figure 2 (a). Figure 2 (b) uses a density threshold (d_{cut}) of one. For each entropy value, we calculate the total volume of mature object-sizes, types, and sites whose entropy is less than or equal to that value. An entropy of zero means uniformity, i.e., all mature objects allocated by the site exhibit the same write behavior. An entropy of one means the site is difficult to predict because objects have a range of write characteristics.

Consider Figure 2 (a) with the results for a w_t of 1K. 100% homogeneity would mean that for that metric, all mature object volume allocated by that site are either write-intensive or not. We observe that 30% of the heap volume are objects with sizes that exhibit uniform write behavior. Using types, 50% of the heap volume has uniform write behavior. Site is a better predictor compared to both size and type — 85% of the mature heap volume originates from sites that are 100% uniform. Surprisingly, size exhibits better write homogeneity than type. For 98.7% homogeneity, both size and type are good predictors. However, site is preferable to size for predicting write intensity because site divides heap memory into more finer-granularity groups. Table 1 shows the number of unique sizes, types and sites for the mature objects in our benchmarks. Note that the number of bytes per site is lower compared to bytes per size.

From Figure 2 (a), we conclude that allocation site is a better predictor of which mature objects get at least 1K writes. We try different write-intensity thresholds and observe similar behavior. As the entropy increases (and homogeneity decreases), the heap volume increases sharply until a point, after which it starts to flatten out in all three curves. More than 90% of the mature heap volume originates from allocation-sites that have 90% of the objects of one kind, write-intensive or non-write-intensive.

Figure 2 (b) shows write density is a better predictor than write frequency. The site homogeneity with a d_{cut} of one is high. Close to 90% of the mature heap volume has 100% of objects either write intensive or not. Size and type are not as predictive. The percentage of homogeneous objects is 40% when considering size, and 50% when considering types.

5.2 Allocation-Site Classification

Although our system promotes mature objects from DRAM to either DRAM or NVM during nursery collections, the system identifies, at allocation time, objects as destined for DRAM or NVM promotions.

We propose two heuristics to classify allocation sites as DRAM or NVM. Both use the homogeneity threshold (h_t) for classifying sites. If the fraction of write-intensive objects allocated from a site is above the homogeneity threshold, then the site is classified as DRAM, otherwise the site is classified as NVM. The heuristics differ in their criteria to identify write-intensive objects.

Write-Frequency (FREQ) FREQ uses the frequency of writes to objects to identify write-intensive objects. If an object gets more than a threshold of writes (write-frequency threshold or w_f), FREQ considers the object as write intensive.

Write-Density (DENS) A drawback of FREQ is that it does not take the size of an object into account. Our purpose is to exploit NVM's capacity and minimize writes to it. We also explore write-density for classifying sites. Write-density of an object is the ratio of writes to the size of an object in bytes. We classify objects with density greater than a threshold (d_{cut}) as write-intensive.

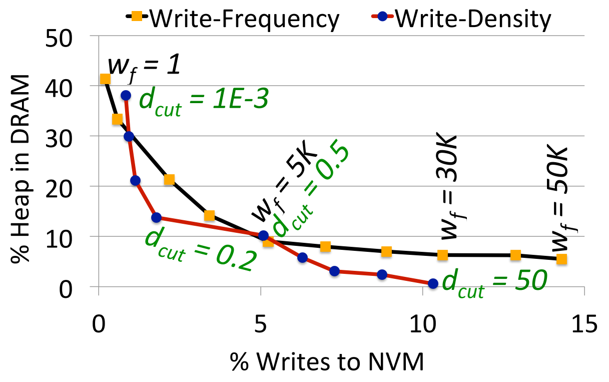


Figure 3: NVM writes and DRAM utilization with FREQ and DENS. ***DENS maximizes NVM's capacity usage while eliminating writes to it.***

6 PREDICTION-GUIDED GARBAGE COLLECTION

Figure 1 shows the basic organization of our approach. Large objects go directly to a non-moving space in DRAM or NVM based on a prediction. The nursery for all other initial object allocations resides in DRAM. Nursery collections promote objects into either DRAM or NVM based on a prediction.

We store the identifiers of the sites classified as write-intensive (DRAM) in an advice file. The advice file is materialized in the runtime system as a hash table indexed by allocation site number. The hash table only contains the write-intensive allocation sites for DRAM. The allocator and collector use this advice for placing objects in hybrid memory.

Allocation. When the mutator allocates objects in the DRAM nursery space, it uses the allocation site identifier to query the hash table. If the allocation site is in the hash table, it marks a bit in the object's header, which indicates that should the collector promotes this object in the future, it should copy it to DRAM.

For large object allocations, the mutator queries the hash table and then directly allocates the large object in either DRAM or NVM, as indicated.

Minor collections. During a nursery collection, as the garbage collector traces the live objects, it checks their write-intensity header bit. If the bit is set, the collector copies the object to the mature DRAM space. Otherwise, it allocates the object in NVM.

Major collections. During a major collection, the garbage collector reclaims dead memory in the DRAM and NVM portions of the mature space. Our system does not relocate mature objects during a major collection because relocation incurs writes. The generous capacity of NVM makes fragmentation less of a concern, but balancing fragmentation and writes could be an area for future work.

7 PLACEMENT RESULTS

This section examines the fraction of writes and utilization of DRAM and NVM using per allocation-site placement advice. We compare the two classification heuristics and show sensitivity to various thresholds. Future systems will likely have orders of magnitude more NVM than DRAM. We therefore explore how writes to

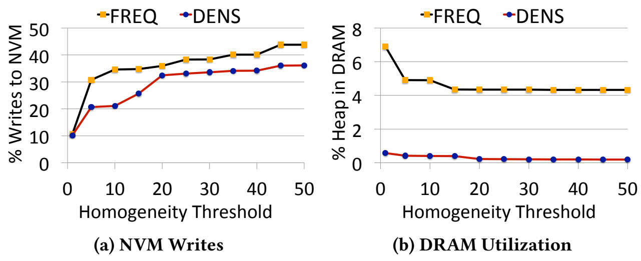


Figure 4: Sensitivity of FREQ and DENS to the homogeneity threshold. ***Low homogeneity thresholds results in fewer writes. High thresholds minimize the DRAM utilization.***

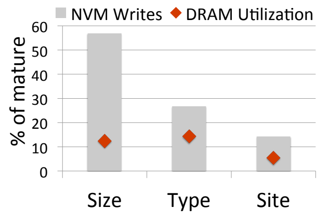
NVM will increase as we minimize the use of DRAM. We measure writes using write barriers for our architecture-independent study.

Figure 3 compares FREQ and DENS in terms of writes to NVM and DRAM utilization averaged across all benchmarks. We set the homogeneity threshold to 1%. We vary the write-frequency threshold in case of FREQ and cutoff density in case of DENS to plot the different points in the figure. We show Pareto optimal tradeoffs in the figure. We observe that FREQ eliminates 99% of the writes to NVM, but places 40% of the heap in DRAM. Both FREQ and DENS result in a lower DRAM utilization if more writes to NVM can be tolerated. With 5% of the writes happening to the NVM mature space, both FREQ and DENS consume the same amount of DRAM. However, DENS results in a higher reduction in DRAM utilization for the same number of NVM writes in most cases. DENS takes into account the writes per byte to objects allocated from a site. With 10% of the writes to NVM, FREQ places 6.3% of the mature heap in DRAM, whereas DENS places only 0.5% of the heap in DRAM, a factor of 13X reduction in DRAM consumption.

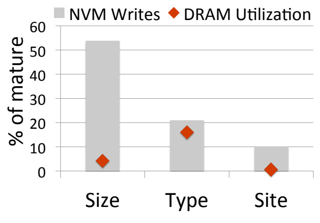
Next, we show the sensitivity of FREQ and DENS to the homogeneity threshold. Figure 4 shows the writes to NVM (a) and the DRAM utilization (b) as the homogeneity threshold increases from 1% to 50%. We configure FREQ and DENS with an w_f of 30 K and d_{cut} of 50 respectively.

Lower homogeneity thresholds minimize writes to NVM, and the least number of writes is obtained using an h_t of 1%. But this results in a high DRAM utilization. Writes to NVM increase with increasing h_t until they plateau out around the 20% threshold. The change in the DRAM utilization shows a different trend. For FREQ, from an h_t of 1% to 5%, there is a sharp drop from 7% to 5%, but after that, the DRAM utilization flattens out. We suspect this is because the sites classified as NVM instead of DRAM with higher homogeneity thresholds occupy a small fraction of the mature heap.

Finally, we compare the site-based predictor to a size-based and a type-based predictor. Similar to site classification, we use FREQ and DENS to classify sizes and types. Figure 5 shows the results. We configure w_f and d_{cut} similar to Figure 4. Using the size-based predictor always results in the largest number of NVM writes. With both FREQ and DENS, using size to predict write-intensive objects leads to more than 50% of the writes to the NVM mature heap. High numbers of writes adversely affect NVM lifetime. Using type is better than size, but site results in the least number of writes to NVM. DRAM utilization with a size-based predictor is less compared to a type-based predictor, but the site-based predictor results in the



(a) Write-Frequency



(b) Write-Density

Figure 5: Comparison of size, type, and allocation-site predictors. *Using an allocation-site predictor eliminates more writes to NVM and exploits the NVM capacity better.*

least amount of DRAM utilization. Future work could consider combining them. Overall, using a site predictor together with DENS eliminates 90% of the writes to NVM, while placing only 0.56% of the heap in DRAM.

8 CONCLUSIONS

This paper contributes a write-intensity predictor for Java applications. The predictor exploits profiles of object writes on a per allocation-site basis. The garbage collector uses the predictor to place objects in DRAM and NVM in a hybrid memory system. We propose two heuristics for site classification. Both require allocation-site homogeneity. FREQ identifies write-intensive objects based on the frequency of writes to objects. DENS uses write-density to identify write-intensive objects and results in a better utilization of DRAM. In particular, DENS eliminates 90% of writes to NVM by placing less than 1% of the mature heap in DRAM. This work requires no modifications to the programming model and is compatible with hardware approaches such as wear leveling, thus having the potential to ease the transition to hybrid memory systems.

ACKNOWLEDGEMENTS

This research is supported through FWO grants no. G.0434.16N and G.0144.17N, European Research Council (ERC) Advanced Grant agreement no. 741097, and VUB's VLAIO Smile-IT project.

REFERENCES

[1] 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).
[2] David A. Barrett and Benjamin G. Zorn. 1993. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*
[3] 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).*
[4] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *International Conference on Software Engineering (ICSE).*
[5] 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 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA).*
[6] Stephen M. Blackburn, Matthew Hertz, Kathryn S. Mckinley, J. Eliot B. Moss, and Ting Yang. 2007. Profile-based Pretenuring. *ACM Trans. Program. Lang. Syst.* 29, 1 (Jan. 2007).
[7] Stephen M Blackburn, Martin Hirzel, Robin Garner, and Darko Stefanović. 2010. pjbb2005: The pseudojbb benchmark, 2005. http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005
[8] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinely, and J. Eliot B. Moss. 2001. Pretenuring for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA).*
[9] Geoffrey W. Burr, Matthew J. Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, BÃŒlent Kurdi, Chung Lam, Luis A. Lastras, Alvaro Padilla, Bipin Rajendran, Simone Raoux, and Rohit S. Shenoy. 2010. Phase change memory technology. *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena* 28, 2 (2010).
[10] Perry Cheng, Robert Harper, and Peter Lee. 1998. Generational Stack Collection and Profile-driven Pretenuring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*
[11] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP).*
[12] 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 *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE).*
[13] 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).*
[14] ITRS. 2015. Internatial Technology Roadmap for Semiconductors 2.0: Executive Report. (2015).
[15] Mark H. Kryder and Chang Soo Kim. 2009. After Hard Drives — What Comes Next? *IEEE Transactions on Magnetics* 45, 10 (2009).
[16] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA).*
[17] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA).*
[18] Onur Mutlu and Lavanya Subramanian. 2014. Research Problems and Opportunities in Memory Systems. *Supercomputing Frontiers and Innovations* 1, 3 (Oct 2014).
[19] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA).*
[20] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS).*
[21] Matthew L. Seidl and Benjamin G. Zorn. 1998. Segregating Heap Objects by Reference Behavior and Lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*
[22] Steven Swanson and Adrian Caulfield. 2013. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *Computer* 46, 8 (2013).
[23] 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).*
[24] Chenxi Wang, Ting Coa, John Zigman, Fang Lv, Yunquan Zhang, and Xiaobing Feng. 2016. Efficient Management for Hybrid Memory in Managed Language Runtime. In *IFIP International Conference on Network and Parallel Computing (NPC).*
[25] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. 2015. Exploiting Program Semantics to Place Data in Hybrid Memory. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT).*
[26] Wangyuan Zhang and Tao Li. 2009. Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT).*