# Prefetching in Hybrid Main Memory Systems

Subisha V  Varun Gohil  Nisarg Ujjainkar  Manu Awasthi
*Indian Institute of Technology Gandhinagar*  *Ashoka University*

## Abstract

The architecture of main memory has experienced a paradigm shift in recent years, with non volatile memory technologies (NVM) like Phase Change Memory (PCM) being incorporated into the hierarchy at the same level as DRAM. This transformation is being carried out by either splitting the memory address across two or more memory technologies, or using a faster technology with higher lifetimes, typically the DRAM, as a cache for the higher capacity, albeit slower main memory made up of a NVM.

Design of such hybrid architectures remains an active area of research from the perspective of DRAM-as-a-cache design, since DRAM could quickly become the bottleneck, as cache lookups require multiple accesses for reading tag and data. In this paper, we augment the DRAM-as-a-cache model with a novel DRAM cache prefetcher that builds on state of the art Alloy Cache. The new DRAM cache architecture allows for prefetching data at both cacheline and page granularities from the NVM, and as a result, provides upto a maximum of $2\times$ performance improvement over a state of the art baseline.

## 1 Introduction

In recent years, there has been an increasing demand from applications for increased memory capacity. However, DRAM stands at the edge of its scaling capabilities and it is unclear if it can be scaled beyond 10 (or so) nano meters [1].

Emerging, non-volatile memory (NVM) technologies like PCM, STT-RAM etc. are touted to provide high capacity alternatives to DRAM as main memory technologies. The capacity benefits of all such technologies are apparent; most technologies have higher areal density [5, 7] than DRAM. However, simply *replacing* DRAM with a NVM causes additional challenges. Owing to the increased access latencies of these technologies, the average memory access time (AMAT) increases significantly, resulting in unacceptable slowdown in application performance [5]. Naive replacement of DRAM with *any* other NVM can lead to significant degradation in performance of memory intensive applications [5]. Not only that, for a main memory comprised entirely of a NVM, a number of other issues like wear leveling and reducing the detrimental performance effects caused by asymmetric read and write latencies need to be addressed. As a result, unless any of these emerging technologies are able to achieve DRAM-compatible latencies, they will have to be used in conjunction with DRAM as main memories.

In the absence of simple solutions, architects have devised *hybrid* memory hierarchies, utilizing both DRAM and NVMs, in order to provide the best of both worlds by providing low-latency data access via DRAM, and higher capacity via NVM. The two most popular hybrid architectures are split-address space [2] and DRAM-as-a-cache [6, 9]. In the split-address space variant, the physical address space is divided, in *some* proportion, across DRAM and NVM. The system software, typically the operating system, has to explicitly perform data placement between the two disparate memory technologies. However, this requires changes at multiple levels. Multiple types of memory controllers, each pertaining to a memory technology, have to be incorporated onto the CPU. The system software also has to be changed substantially [12].

To make the adoption of hybrid memory hierarchies less intrusive, an alternate architecture which uses DRAM as a cache to the NVM comprised of main memory was proposed [6]. In this design, hardware seamlessly manages data between the DRAM and NVM. The AMAT experienced by the application is closer to that of DRAM, but visible memory capacity is that of NVM.

Multiple architectures for DRAM-as-a-cache have been proposed [8]. Since a cache requires storage of both tag and data, overall access latency could increase if accesses to both are serialized. This becomes especially problematic in the case of DRAM caches, since serializing these accesses requires multiple round trips to DRAM, increasing overall access time, reducing effectiveness of the cache. Architectures like Alloy Cache [9] have been proposed to alleviate these shortcomings of DRAM caches. Alloy Cache allows for reduction of average latency to access DRAM cache by storing

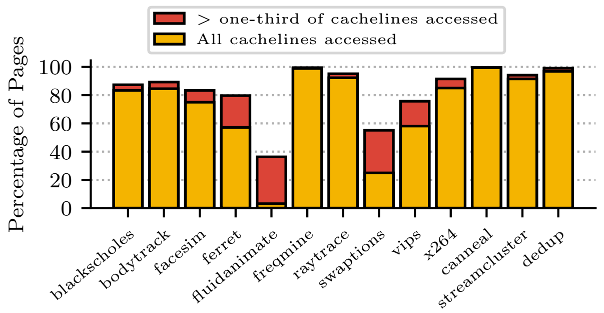


Figure 1: Cacheline accesses of NVM pages

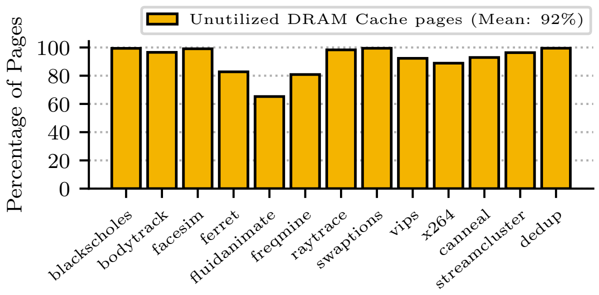


Figure 2: Page Utilization in DRAM Cache

and retrieving tag (8 bytes) and data (64 bytes) together and serving it in a single burst.

We posit that one can further improve the AMAT by intelligently prefetching data from NVM to DRAM Cache. To the best of our knowledge, there exists no work for designing prefetchers for DRAM caches. If If prefetchers for DRAM caches are to be designed, we need to address some basic questions regarding those. For example, what should be the granularity of prefetch? Where should the prefetched data be placed? We try to answer some of these questions in this paper.

To gain insights into the workload behavior under DRAM Cache based hybrid memories, and their suitability for prefetchers, we conduct multiple experiments. First, we analyze per-page cacheline access characteristics of different applications [1]. Assuming a 64 B cacheline and 4 KB pages, each page contains 64 cachelines. Figure 1 shows that on average, *every* cacheline of 73% of the pages is accessed during program's execution. In addition, 84% of the pages have *at least* a third of their cachelines accessed. These experiments provide support for the presence of spatial locality in the workloads under consideration, and the fact that fetching data into the DRAM cache at larger than cacheline granularity will be helpful for application performance.

However, fetching at larger granularities can lead to cache pollution. In order to assess such detrimental effects, we tracked space allocation in Alloy Cache, the results of which are presented in Figure 2. We observe that large amount of DRAM cache capacity remains unutilized - across all workloads, 92% of all DRAM cache pages are unallocated.

These results provide us two insights. First, data in DRAM cache should be managed at multiple granularities, both large and small. Second, managing data at cacheline granularity only leads to under-utilization of DRAM Cache capacity. We combine these two insights to design a novel prefetcher that fetches data at page granularity into regions of DRAM cache that are not utilized by the Alloy Cache.

[1] Section 3 describes our experimental methodology.

## 2 Prefetcher Design

In the baseline Alloy Cache design, all pages in DRAM are used for storing cachelines, brought from NVM due to demand misses. Each such cacheline maps to a unique DRAM page, and each such page might contain cachelines from multiple physical pages in NVM.

However, as described before, many physical pages in NVM exhibit significant spatial locality, and the application will benefit if the entire page, and hence all its cachelines, were prefetched into the DRAM. If this was enabled, the DRAM Cache will now have two kinds of pages:

**Alloy Cache Page:** These are the DRAM Cache pages that consist of Alloy Cachelines. Alloy Cachelines have data (64B) and tag (8 B) integrated together as a single unit (72 B). Hence, one 4KB Alloy Cache page can accommodate 56 cachelines.

**Prefetched Page:** These pages are prefetched from NVM to the DRAM Cache. Tags for cachelines in these pages are saved separately in the prefetcher component, which is described later in Section 2.4. A 4 KB Prefetched Page can accommodate 64 cachelines.

Prefetching physical pages from NVM into DRAM, while keeping the baseline design of Alloy Cache requires us to provide additional functionality to various memory controllers in the hybrid memory architecture, which are described next.

We add the NVM Page Classifier (NPC) to NVM's memory controller that decides if a page should be prefetched. It does so by maintaining the access history for cachelines in recently accessed pages, and prefetching pages with most potential future accesses. To improve DRAM utilization, we allocate prefetched pages to currently unallocated regions in DRAM Cache. This requires augmenting the DRAM Cache's memory controller with an additional structure called the Empty Page Classifier (EPC). EPC keeps track of *empty*, or currently unoccupied, pages. Another structure, called the Type Classifier (TC) is maintained to distinguish accesses to different types (Alloy Cache vs. Prefetched) of pages. We also include a Page Redirection Table (PRT) in DRAM Cache's memory controller. It maintains a mapping of prefetched pages to their corresponding locations in DRAM Cache.

Figure 3 illustrates the high level architecture of the proposed prefetcher and new components that are added to the

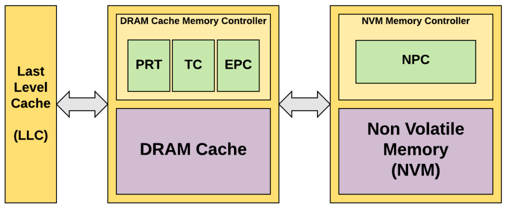


Figure 3: Proposed structures in Memory Controllers

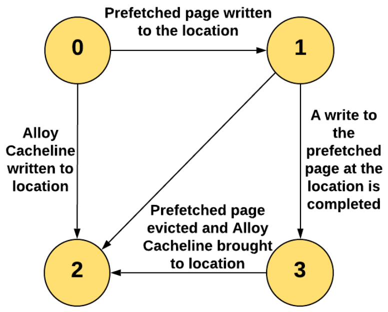

0
1
2
3
Prefetched page written to the location
Alloy Cacheline written to location
A write to the prefetched page at the location is completed
Prefetched page evicted and Alloy Cacheline brought to location

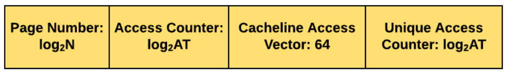
Figure 4: State transition diagram of Type Classifier

memory controllers. The structures we introduce are orthogonal to those of Alloy Cache which allows our prefetcher to work with any other DRAM Cache design.

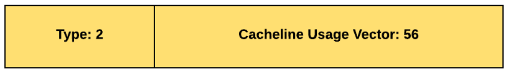
## 2.1 NVM Page Classifier (NPC)

The NVM Page Classifier keeps track of the access history of recently accessed NVM pages, which is used to identify pages suitable for prefetching. Ideally, we should prefetch a page with high number of accesses and with accesses to a number of unique cachelines. To identify such pages, we keep track of the number of cachelines touched in individual NVM pages. An NVM page is deemed a candidate for prefetch when the number of accesses to the page crosses an Access Threshold (AT), and the number of accesses to unique cachelines crosses Unique Access Threshold (UAT). Here, AT $\geq$ UAT.

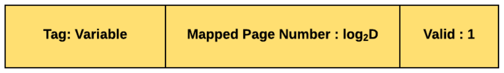
Figure 5a illustrates the composition of a single NPC entry. Considering there are $N$ total NVM pages, each NPC entry has $log_2 N$ bits to identify the page number. An Access Counter keeps track of the number of accesses to this page. We provide a $log_2 AT$ bit saturating counter to keep track of Access Threshold. A 64 bit Cache Access Vector (CAV), with one bit for each cacheline in the page, is also provided. A set bit in CAV indicates that the cacheline corresponding to the bit has been accessed. Finally, we provide a $log_2 AT$ bit saturating counter for keeping track of unique cachelines accessed. Whenever an NVM page is accessed, the access counter is incremented. The corresponding bit in the CAV is checked. If the bit is 0, it is set and the UAT is incremented. The NPC will contain multiple such entries. For every request a fully-associative search is performed on the NPC whose entries are evicted using LRU.


Page Number: $log_2N$
Access Counter: $log_2AT$
Cacheline Access Vector: 64
Unique Access Counter: $log_2AT$

(a) NVM Page Classifier



(b) Type Classifier



(c) Page Redirection Table

Figure 5: Entries of Prefetcher components. Values are the number of bits required

## 2.2 Type Classifier (TC)

As we prefetch pages directly to DRAM Cache, it now houses two types of pages: Alloy Cache Page and Prefetched Page. To use the system without errors, we need to identify the type of page that exists at a DRAM Cache location. The Type Classifier identifies whether a DRAM Cache location houses an Alloy Cache page or a Prefetched page. It also maintains the the occupancy details of all the cachelines in the Alloy Cache page. Figure 5b represents an entry in the TC table. Each TC entry has 2 bits to identify the type of page stored at the location. Using 2 bits, we can represent the following four states:
State 0 : 00 $\leftarrow$ Empty/Un-allocated Location
State 1 : 01 $\leftarrow$ Valid Prefetched Page at Location
State 2 : 10 $\leftarrow$ Valid Alloy Cache Page at Location
State 3 : 11 $\leftarrow$ Dirty Prefetched Page at Location
When a page is prefetched from NVM to DRAM, the page is in state 1. When we write to a page in state 1, it transitions to state 3, indicating that the page is dirty. All Alloy Cache pages are in state 2, irrespective of cachelines being dirty. The dirtiness of an Alloy Cacheline is indicated by the tag stored along with the data. Figure 4 illustrates the transitions between all four states.

Each TC entry contains a Cacheline Usage Vector (CUV), having one bit for each cacheline. These bits are valid only if the Type bits indicate State 2 (Alloy Cache page at location). A 4KB Alloy Cache page has 56 cachelines, implying a 56 bit CUV. A set bit signifies that the corresponding cacheline is occupied. An entry in Type Classifier is maintained for all locations in the DRAM Cache.

## 2.3 Empty Page Classifier (EPC)

The Empty Page Classifier keeps track of empty pages in DRAM Cache. The EPC is used to identify an empty page in DRAM Cache and place a prefetched page at that location, improving DRAM Cache utilization.

Figure 6 represents the organization of the multi-level

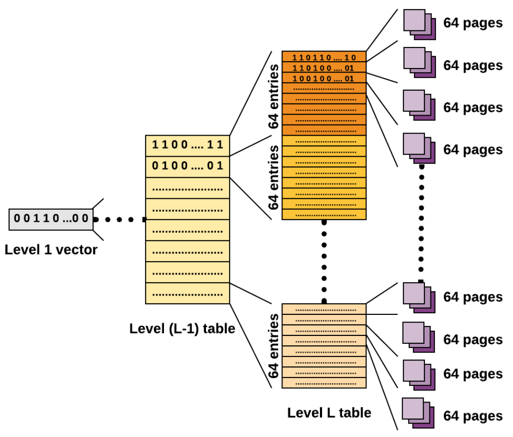


Figure 6: Structure of Empty Page Classifier

EPC table. For implementing the EPC, we group consecutive DRAM pages in groups of 64, each represented by a 64 bit vector. The 64 bit vectors for all groups together form the last Level $L$ of the EPC table. A set bit in this vector indicates that the corresponding page is empty. We further group these vectors in groups of 64, leading to a 64 bit vector in Level $(L-1)$ of the EPC table. We recursively keep grouping the bit vectors in groups of 64 or less, till we have a single vector left i.e. we reach Level 1 of the EPC table. A set bit in the bit vectors from Level 1 to $(L-1)$ indicates that there exists *at least* one set bit in the corresponding bit vector in the next level. Such a multi-level design improves the worst-case lookup latency when compared to a single level design.

The EPC is updated when a DRAM Cache page becomes empty or is occupied. This is done by first updating the last level and then progressing towards upper levels, updating the structure at each step. We perform a lookup on EPC for the last empty page by finding the last set bit in the first level of EPC and recursively traversing through the lower levels till we reach a page. For implementing this, we use the hardware find-first-set (ffs) function, which finds the index of the first set bit starting from the least-significant bit.

## 2.4 Page Redirection Table (PRT)

The Page Redirection Table maintains a mapping of prefetched pages to their locations in DRAM Cache, and has a functionality similar to that of a page table. Every miss in the last level cache first accesses the PRT to identify whether the requested address exists as a cacheline in a prefetched page.

The PRT is a hash-table. We empirically find the optimal size and associativity for the PRT which minimizes collisions and improves look-up time. Figure 5c depicts an entry in PRT. Each PRT entry stores the tag of the cacheline to verify that the the cacheline in the prefetched page is indeed the requested cacheline. The number of tag bits depend of the number of offset bits, index bits and data bits in the address. To specify the location in DRAM Cache where the page is mapped to, we have $log_2 D$ bits in each PRT entry. We also have a valid bit which when set indicates that the mapping is valid. A PRT entry is invalidated when an Alloy Cacheline is brought to the location which is occupied by a prefetched page due to some request. As demand fetches have higher priority than prefetches, we evict prefetched page and invalidate the PRT entry. We use LRU policy for evicting entries from PRT.

## 2.5 Memory Request Service Routines

Algorithm 1 and 2 provide the service routines for write and read requests, respectively. Next, we cover the salient points of the read/write routines, including some interesting cases.

Given our design, there is a chance that a requested cacheline might be present at two locations in the DRAM: as a part of a Prefetched Page, which contains all the cachelines for the page. Or, it can be present in an Alloy Cache page, which might contain multiple other cachelines. The case for writing data when it is present in either of the two locations is easy. However, care has to be taken to ensure data consistency when a cacheline is present at multiple locations.

To ascertain the cacheline's presence, we first read the PRT and TC in parallel. A *PRT hit* indicates that the cacheline is present in a Prefetched Page. If there is a hit in the Alloy Cache as well, it indicates that the cacheline is also present as an Alloy Cacheline.

When requested cacheline is present in both, a prefetched page and an Alloy Cacheline, the read request is serviced from the Alloy Cacheline. In parallel, if the cacheline is dirty, the data is written to the corresponding region in the Prefetched Page and the cacheline is invalidated. In the case of a write request, the data is written to the cacheline in the Prefetched Page. If the Alloy Cacheline was dirty, the Prefetched Page is updated with the dirty cacheline before the current write is carried out. The Alloy Cacheline is also invalidated.

In cases of read requests where the requested cacheline is present in a Prefetched Page, and the corresponding Alloy Cache page doesn't have the requested cacheline, the data is returned from the Prefetched Page. The writes are also done at the corresponding location in the Prefetched Page.

Finally, there can be write cases where the cacheline needs to be written into the DRAM Cache, and is not present in any Prefetched Page. If the cacheline is present at its (fixed) location as an Alloy Cacheline, it is written in place. In case this location is currently being occupied by a Prefetched Page, the page will be evicted, and it's contents written back to NVM, if any cachelines are dirty. The location is now marked as an Alloy Cache page, and the write to the cacheline completes.

In cases where data is not present in either in a Prefetched Page, or as a part of a Alloy Cache page, the request for data is sent to the NVM. The data is then brought in at cacheline granularity, unless the criteria for prefetching the page are

**Algorithm 1:** Write Request Service Routine

```
input  : Address and Data
if PRT hits for Address then
    if TC is 2 then
        if Cacheline hits then
            Invalidate Cacheline
        end
    end
    Write Data to prefetched page
else
    if TC is 1 then
        Evict prefetched page at DRAM Cache location
    end
    if TC is 3 then
        Evict prefetched page at DRAM Cache location and write evicted
          page to the NVM.
    end
    if TC is 2 then
        if Cacheline does not hit then
            Evict the Cacheline at the DRAM Cache location.
        end
    end
    write Data to Cacheline at DRAM Cache location.
end
```

fulfilled by this request.

## 3 Experimental Setup

We use ZSIM [10] and NVMain [4] for simulating hybrid main memory hierarchy with 1 GB 8-channel DRAM Cache and 16 GB Phase Change Memory (PCM). We chose PCM as a representative NVM technology, but the same set of experiments can be repeated by changing the NVMain parameters to match those of the memory technology that should be modeled. We modify the source code to incorporate our proposed prefetcher. We simulate an 8 core, 2.6 GHz processor with private L1 instruction, L1 data and L2 caches for each core. The L3 cache is shared across all cores. Table 1 shows the cache hierarchy and main memory configuration used for our experiments. We assume a 4 KB page size for a total of 4,194,304 pages in the NVM and 2,62,144 pages in DRAM Cache. As a result, we need 22 bits to represent NVM page number and 18 bits to represent DRAM Cache page number.

We observe that 82% of pages with accesses to at least one-third of their cachelines, have all their cachelines accessed (Figure 1). Hence, we set AT to 22 ( $\approx$ 64/3). We empirically determine that it is best to set UAT as two-thirds of AT. Hence we set UAT to 15 ($\approx$ AT*2/3). Each NPC entry is 12 bytes with 22 pagenumber bits, 5 bits each for Access Counter and Unique Access Counter, and 64 bits for Cacheline Access Vector. Further, we find that the optimal size of NPC is 16 entries, bringing the total NPC size to 192 bytes. We assume a 1 clock cycle overhead for accessing the NPC.

To cover all pages in the DRAM Cache we require a 2MB storage space for the TC table, shown in Figure 5b . Using CACTI [11], the access latency of TC table is estimated to be 4 clock cycles.

From design space exploration, we find that it is best to have a 4 way set associative table with 1024 sets for the PRT.

**Algorithm 2:** Read Request Service Routine

```
input  : Address to be read
if PRT hit occours for Address then
    if TC is 0 or 1 or 3 then
        return data from prefetched page
    else
        if Cacheline hits and Cacheline is dirty then
            Write DRAM Cacheline to prefetched page.
            Invalidate the DRAM Cacheline.
        end
        return data from prefetched page
    end
else
    if TC is 0 then
        Send read request to NVM
    else if TC is 1 then
        Evict prefetched page at DRAM Cache location
        Send read request to NVM
    else if TC is 2 then
        if Cacheline hits then
            return data from DRAM Cacheline
        else
            Evict cacheline at DRAM Cache location.
            If evicted cacheline is dirty and belongs to a prefetched
              page, write the evicted cacheline to that prefetched page.
            Send read request to NVM
        end
    else
        Evict prefetched page at DRAM Cache location and write evicted
          page to the NVM.
        Send read request to NVM
    end
end
```

Table 1: Experimental Configuration

**Cache Hierarchy Configuration**

| | Size (KB) | Latency (Cycles) |
|---|---|---|
| **L1 Insn** | 32 | 4 |
| **L1 Data** | 32 | 4 |
| **L2/L3 Unified** | 256/4096 | 8/16 |

*All caches are 8 way set associative*

**Main Memory Configuration**

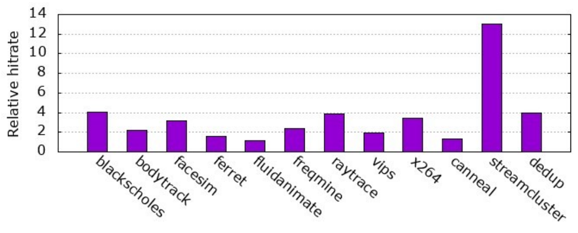
| | DRAM Cache | PCM |
|---|---|---|
| **Frequency (MHz)** | 1600 | 400 |
| **tRCD/tCCD (cycles)** | 23/4 | 312/13 |
| **tCAS/tRP (cycles)** | 23/23 | 7/390 |

This table is 20KBs, with a single entry of 5 bytes. While accessing the PRT, we use 10 bits (Bits 22-13) to index into the set and last 12 bits for offset. For our configuration, each PRT entry has 21 tag bits, 18 bits for DRAM Cache page number and 1 valid bit. We determine the access latency of PRT to be 2 clock cycles using CACTI [11].

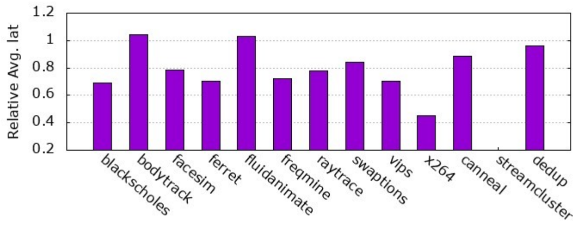
We implement the EPC on a per-channel basis. Our configuration, has 32,768 pages per channel, leading to a 3 level EPC table described in Section 2.3. EPC's Level 3 has 512 64-bit vectors, Level 2 of EPC has 8 64-bit vectors and Level 1 is a 8-bit vector. Overall, the EPC is approximately 33 KBs and has a 3 cycle latency, as per CACTI [11].

## 4 Results

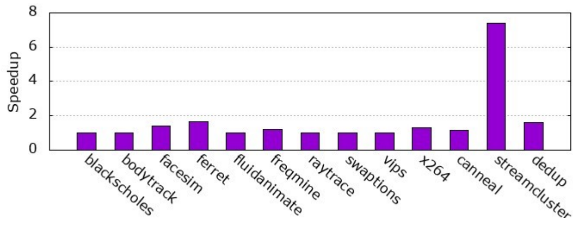
We evaluate the prefetcher on the PARSEC [3] benchmarks with simlarge input set, for 2 cases - one being single-program workloads in which we run a single thread of each application for 5 billion instructions. The other is for multi-programed workloads where two instances of a single threaded applica-



(a) DRAM Cache hitrate relative to Alloy Cache



(b) Avg. memory latency relative to Alloy Cache



(c) Speedup relative to Alloy Cache

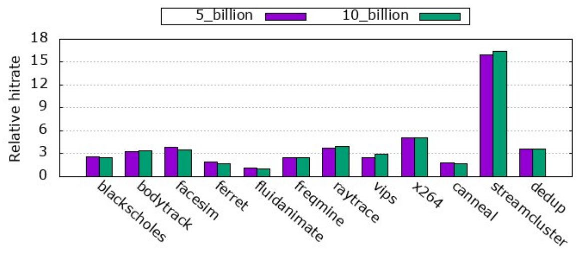
Figure 7: Results for Single-Program workloads

tion are simultaneously executed for both 5 and 10 billion instructions.

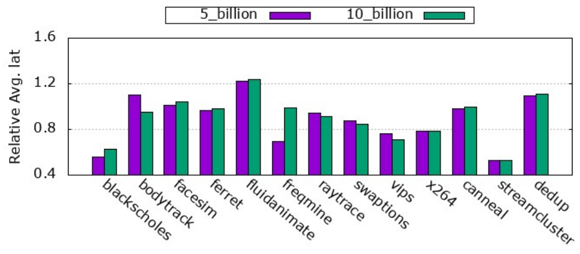
Figure 7a shows the cache hit rate improvement for single-program workloads. We observe a 1.5-4× improvement in DRAM Cache hit rate across all PARSEC workloads. Streamcluster experiences a 12× improvement in cache hit rate. We believe that this is due to streamcluster's memory intensive behavior – no other application in the PARSEC suite generates more than 50% of streamcluster's DRAM Cache requests.

The improvement in cache hit rate also translates to lowered AMAT, shown in Figure 7b. Average memory access time for streamcluster reduces by 80%. For other applications we observe a 10-30% decrease in average memory latency. However, for bodytrack and fluidanimate, average memory access latency increases slightly despite increase in DRAM Cache hit-rate. We believe that this is due to the overheads introduced by prefetcher components and the lack of memory level parallelism of these applications.

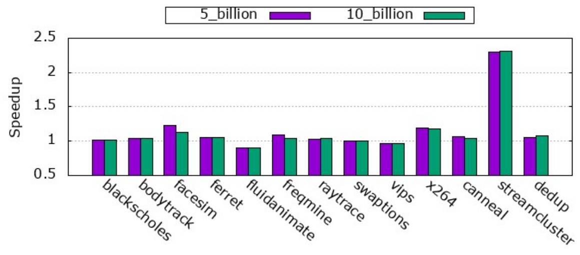
Further, Figure 7c depicts the speedup of the prefetcher working in conjunction with Alloy Cache over the baseline Alloy Cache design. We use the ratio of instructions-per-cycle (IPC) as a measure of speedup. For streamcluster we observe a 7× speedup, while for fluidanimate and vips we do not see any performance impact. The rest of the applications observe 16-40% improvements in IPC.



(a) DRAM Cache hitrate relative to Alloy Cache



(b) Avg. memory latency relative to Alloy Cache



(c) Speedup relative to Alloy Cache

Figure 8: Results for Multi-Program workloads

Figure 8 shows the results for multi-program workloads. Results are obtained by concurrently running 2 single-threaded instances of the same program. Two types of runs were done, one for 5 billion instructions, and the other for 10 billion. We observe a 2× increase in DRAM Cache hit rate, on average, across all workloads except streamcluster, for both 5 and 10 billion instructions. For streamcluster we observe a 16× improvement in the hit rate. Streamcluster, owing to a 50% reduction in AMAT, experiences a 2× speedup over the baseline Alloy Cache. The rest of the workloads experience a 10-40% reduction in AMAT.

## 5 Conclusion

In this paper, we provide the design and initial set of results for a novel prefetcher for the DRAM Cache in hybrid memory architectures. The prefetcher brings data at the granularity of pages from NVM in conjunction with Alloy Cache, which performs demand fetching at cacheline granularity. We show that it is possible to co-locate cached data with both cacheline and page granularities by augmenting the DRAM memory controller with a few low-latency and low overhead structures. Finally, we demonstrate that the novel prefetcher design has the potential to outperform the state of the art Alloy Cache baseline by up to 2× for memory intensive workloads.

## Acknowledgements

We thank Biswabandan Panda, IIT Kanpur for initial discussions and our shepherd Deepavali Bhagwat, IBM Research for her help in improving the manuscript.

## References

[1] DRAM Scaling Challenges Grow. https://semiengineering.com/dram-scaling-challenges-grow/. Accessed: 21-03-2020.

[2] Frank Bellosa. When physical is not real enough. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, page 25–es, New York, NY, USA, 2004. Association for Computing Machinery.

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[4] Asif Ali Khan, Fazal Hameed, and Jeronimo Castrillon. Nvmain extension for multi-level cache systems. In *Proceedings of the Rapido'18 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, pages 1–6, 2018.

[5] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 2–13, New York, NY, USA, 2009. Association for Computing Machinery.

[6] Gabriel H. Loh and Mark D. Hill. 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-44, page 454–464, New York, NY, USA, 2011. Association for Computing Machinery.

[7] Darsen Lu. Tutorial on emerging memory devices. *Available at people. oregonstate. edu/~sllu/Micro_MT/presentations/micro16_emerging_mem_tutorial_darsen. pdf*, 2016.

[8] Sparsh Mittal and Jeffrey S Vetter. A survey of techniques for architecting dram caches. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1852–1863, 2015.

[9] Moin Qureshi and Gabriel H Loh. Fundamental latency trade-offs in architecturing dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proc. of the 45th Intl. Symp. on Microarchitecture, Vancouver, Canada*, volume 10, 2012.

[10] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news*, 41(3):475–486, 2013.

[11] Shyamkumar Thoziyoor, N Muralimanohar, J Ahn, and N Jouppi. Cacti 6.5. *hpl. hp. com*, 2009.

[12] Zi Yan, Ján Veselý, Guilherme Cox, and Abhishek Bhattacharjee. Hardware translation coherence for virtualized systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 430–443, New York, NY, USA, 2017. Association for Computing Machinery.