Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation

Xiangyao Yu[1] Christopher J. Hughes[2] Nadathur Satish[2] Onur Mutlu[3] Srinivas Devadas[1]
[1]MIT [2]Intel Labs [3]ETH Zürich

ABSTRACT

Placing the DRAM in the same package as a processor enables several times higher memory bandwidth than conventional off-package DRAM. Yet, the latency of in-package DRAM is not appreciably lower than that of off-package DRAM. A promising use of in-package DRAM is as a large cache. Unfortunately, most previous DRAM cache designs optimize mainly for cache hit latency and do not consider bandwidth efficiency as a first-class design constraint. Hence, as we show in this paper, these designs are suboptimal for use with in-package DRAM.

We propose a new DRAM cache design, *Banshee*, that optimizes for both in-package and off-package DRAM bandwidth efficiency without degrading access latency. Banshee is based on two key ideas. First, it eliminates the tag lookup overhead by tracking the contents of the DRAM cache using TLBs and page table entries, which is efficiently enabled by a new lightweight TLB coherence protocol we introduce. Second, it reduces unnecessary DRAM cache replacement traffic with a new bandwidth-aware frequency-based replacement policy. Our evaluations show that Banshee significantly improves performance (15% on average) and reduces DRAM traffic (35.8% on average) over the best-previous latency-optimized DRAM cache design.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; *Heterogeneous (hybrid) systems*;

KEYWORDS

DRAM Cache, Main Memory, In-Package DRAM, Hybrid Memory Systems, TLB Coherence, Cache Replacement

ACM Reference format:
Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, Onur Mutlu, Srinivas Devadas. 2017. Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation. In *Proceedings of The 50th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, MA, USA, October 14-18, 2017 (MICRO-50),* 14 pages.
https://doi.org/10.1145/3123939.3124555

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO-50, October 14-18, 2017, Cambridge, MA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3124555

1 INTRODUCTION

In-package DRAM technology integrates a CPU and a high-capacity DRAM in the same package, enabling much higher main memory bandwidth to the CPU than traditional off-package DRAM. For memory bandwidth-bound applications (e.g., graph processing, some machine learning algorithms, sparse linear algebra-based HPC codes), in-package DRAM can significantly boost system performance [9, 10, 17, 30, 34]. Several hardware vendors are offering processors with in-package DRAM (e.g., Intel's Knights Landing [56], AMD's Fiji [3], and Nvidia's Pascal [2]) and a large number of systems have been designed to take advantage of in-package DRAM [18, 20, 27, 32, 38, 39, 41, 45, 50, 54, 59].

One critical property of in-package DRAM is that, while it provides high bandwidth, *its latency is similar to or even worse than off-package DRAM* [1, 19, 55]. This is because the throughput computing applications that modern in-package DRAM products target are typically latency-tolerant, but bandwidth-intensive. Many previous DRAM cache designs, however, assume low-latency in-package DRAM and therefore are not necessarily the best fit for practical systems with in-package DRAM.

In particular, many of the existing DRAM cache designs incur large amounts of traffic to in-package and/or off-package DRAM for *metadata management* (e.g., fetching DRAM cache tags) and *cache replacement*. Since the tag array of a multi-gigabyte cache cannot easily fit in on-die SRAM, many previous designs (especially *fine-granularity* designs)[1] store the tag array in the in-package DRAM itself [32, 45, 50], which is accessed for each tag lookup. Although the latency of a tag lookup can be largely hidden using previously proposed techniques [32, 45, 50], the metadata accesses consume valuable bandwidth. To reduce metadata traffic, previous works [28, 38] propose to manage data at *page-granularity* and track the contents of the DRAM cache using the Page Table Entries (PTEs) and Translation Lookaside Buffers (TLBs). Tag lookups become essentially free in terms of latency via this page-table-based mapping mechanism. Unfortunately, these existing designs incur considerable complexity and performance overhead for maintaining coherent address mappings in TLBs across all the cores.

For page-granularity DRAM cache designs, cache replacement can incur excessive DRAM traffic because each replacement moves a full page of data between in-package and off-package DRAM. For pages with little spatial locality, most of the transfered data is never accessed by the processor, wasting both in-package and off-package DRAM bandwidth. The problem becomes even worse when large pages (e.g., 2MB or 1GB) are used in the system. Furthermore, accessing and updating the metadata (e.g., the LRU bits or the

[1]By a fine-granularity design, we refer to a DRAM cache design with a block size that is equal to the block size of the processor's L1/L2 caches (e.g., 64 bytes). A page-granularity DRAM cache, in contrast, has a block size that is the same as a virtual memory page (e.g., 4 KB).

frequency counters) also incurs extra DRAM traffic if the metadata is stored in the DRAM cache itself. Existing solutions, like the footprint cache [28, 31], can mitigate the bandwidth pressure to some extent, by caching only the parts of a page that will likely be accessed by the processor. However, our evaluation (Section 6) shows that significant room for improvement still remains.

In this paper, we propose *Banshee*, a DRAM cache design whose goal is to maximize the bandwidth efficiency of both in-package and off-package DRAM, while also providing low access latency. Banshee is based on two key ideas.

First, similar to previous work [38], Banshee avoids tag lookups by tracking the DRAM cache contents using the PTEs and TLB entries. Different from previous work, however, Banshee uses a novel, lightweight TLB coherence mechanism that has low complexity. Specifically, Banshee maintains a small hardware table (called the Tag Buffer) at each memory controller that stores information about whether or not a page is cached, for recently inserted or replaced pages. The PTEs and TLB entries are *lazily* updated *only* when the Tag Buffer is full, which greatly amortizes the performance overhead of TLB coherence.

The second key idea in Banshee is a *bandwidth-efficient frequency-based replacement* (FBR) policy, whose goal is to reduce unnecessary DRAM cache replacement traffic. To this end, Banshee reduces 1) the number of replacements in an FBR policy by limiting the rate at which replacements happen, via a *bandwidth-aware* FBR policy, 2) the amount of metadata traffic (i.e., reads and updates to frequency counters), by accessing the metadata for only a *sampled* fraction of memory accesses.

Specifically, this work makes the following contributions:

(1) We propose a *lazy*, lightweight TLB coherence mechanism, which is simpler and more efficient than the TLB coherence mechanism in previous page-table-based DRAM cache designs.
(2) We propose a new bandwidth-aware frequency-based replacement policy, which significantly improves DRAM bandwidth efficiency by minimizing unnecessary data and metadata movement in a page-granularity DRAM cache.
(3) By combining page-table-based page mapping management and bandwidth-efficient cache replacement, Banshee significantly improves in-package DRAM bandwidth efficiency. Compared to three other state-of-the-art DRAM cache designs, Banshee outperforms the best of them (Alloy Cache [50]) by 15.0% while reducing in-package DRAM traffic by 35.8%.

2 BACKGROUND

Data set sizes are increasing in application domains like big data analytics, machine learning, and high performance computing. For these applications, memory bandwidth can be a primary performance bottleneck. To meet the high memory bandwidth requirements of these applications, in-package DRAM have been built into both CPUs and GPUs, and, in some existing systems, are managed as caches [2, 56].

The bandwidth of an in-package DRAM should be used judiciously for maximal performance. This is because some carefully-optimized applications can fully utilize the in-package DRAM bandwidth such that transferring data that is *not* used by the application (e.g., DRAM cache metadata, extra traffic caused by excessive cache replacement) can limit system performance.

While in-package DRAM bandwidth is growing, so is the compute capability of a chip. We do not expect the bandwidth/compute ratio to change drastically in the near future. Therefore, improving the bandwidth efficiency of DRAM caches will be beneficial for overall system performance. To make this more concrete, we computed the Bandwidth/FLOPS[2] ratio for Nvidia's P100 (Pascal) system (0.14 B/FLOP) [4], Nvidia's V100 (Volta) system (0.12 B/FLOP) [6], and Intel's Knights Landing system (0.12 B/FLOP) [56]. We found them to be very similar, even though these three systems' absolute in-package DRAM bandwidth is different by 2×.[3] This shows that compute power increases as additional DRAM bandwidth is available.

In this section, we discuss the design space of DRAM caches, and where previous proposals fit in that space. We show the bandwidth inefficiency of previous schemes with respect to two major design considerations: 1) tracking the cache contents, i.e., mapping (Section 2.1), 2) changing the cache contents, i.e., replacement (Section 2.2).

For our discussion, we assume, without loss of generality, that the processor has an SRAM last-level cache (LLC) managed at fine (64 B) granularity.[1] Physical addresses are mapped to memory controllers (MC) statically at coarse (4 KB page) granularity. We also assume the in-package DRAM is similar to the first-generation High Bandwidth Memory (HBM) [29, 46]. The link width between the memory controller and HBM is 16B, but with a minimum data transfer size of 32B. Thus, reading a 64B cacheline plus the tag requires the transfer of at least 96B. We also assume the in-package and off-package DRAM have the same latency [55].

2.1 Tracking DRAM Cache Contents

For each LLC miss, the memory controller determines whether to access the in-package or the off-package DRAM. Therefore, the mapping of where each data block resides must be stored somewhere in the system. Mapping can be managed either using tags or through the virtual-to-physical address remapping.

2.1.1 Using Tags. The most common technique for tracking the contents of a cache is explicitly storing the set of tags, i.e., the bits from each data block's address to uniquely identify the block. However, the tag storage can be significant when the DRAM cache is large. A 16 GB DRAM cache, for example, requires 512 MB (or 8 MB) tag storage if managed at fine (or coarse) granularity.[1] As a result, many state-of-the-art DRAM cache designs store tags in the in-package DRAM itself. These designs consume extra DRAM bandwidth for tag lookups associated with DRAM cache accesses.

Table 1 summarizes the characteristics of some state-of-the-art DRAM cache designs, including two that store tags in the in-package DRAM, *Alloy Cache* [50] and *Unison Cache* [32].

[2] *FLOPS* is the number of *floating point operations per second* that can be supported by the processor.
[3] P100 [4] has 16GB in-package High Bandwidth Memory (HBM) with a bandwidth of 732 GB/s. Its cores provide 5300 GFLOPS for double-precision operations. V100 [6] has 16 GB of in-package HBM2 with a bandwidth of 900 GB/s. Its cores provide 7500 GFLOPS for double-precision operations. Intel Xeon Phi 7290 [56] has 16 GB in-package DRAM with bandwidth of 400+GB/s. Its cores provide 3456 GFLOPS for double-precision operations.

Table 1: Summary of Operational Characteristics of Different State-of-the-Art DRAM Cache Designs – We assume perfect way prediction for Unison Cache. Latency is relative to the access time of the off-package DRAM (see Section 6 for baseline latencies). We use different colors to indicate the high (dark red), medium (white), and low (light green) overhead of a characteristic.

Scheme	DRAM Cache Hit	DRAM Cache Miss	Replacement Traffic	Replacement Decision	Large Page Caching
Unison [32]	In-package traffic: 128 B (data + tag read and update) Latency: ~1x	In-package traffic: 96 B (spec. data + tag read) Latency: ~2x	On every miss Footprint size [31]	Hardware managed, set-associative, LRU	Yes
Alloy [50]	In-package traffic: 96 B (data + tag read) Latency: ~1x	In-package traffic: 96 B (spec. data + tag read) Latency: ~2x	On some misses Cacheline size (64 B)	Hardware managed, direct-mapped, stochastic [20]	Yes
TDC [38]	In-package traffic: 64 B Latency: ~1x TLB coherence	In-package traffic: 0 B Latency: ~1x TLB coherence	On every miss Footprint size [28]	Hardware managed, fully-associative, FIFO	No
HMA [44]	In-package traffic: 64 B Latency: ~1x	In-package traffic: 0 B Latency: ~1x	Software managed, high replacement cost		Yes
Banshee (This work)	In-package traffic: 64 B Latency: ~1x	In-package traffic: 0 B Latency: ~1x	Only for hot pages Page size (4 KB)	Hardware managed, set-associative, frequency based	Yes

Alloy Cache [50] is a direct-mapped DRAM cache that stores data at fine granularity. The tag and data for a set are stored adjacently in the DRAM cache. For every tag probe, the data is speculatively loaded as well. Thereby, on a cache hit, the latency is roughly that of a single DRAM access. On a cache miss, Alloy Cache incurs latency of an in-package DRAM access *plus* an off-package DRAM access. In terms of bandwidth consumption, Alloy Cache 1) loads the tag and data from DRAM cache, 2) loads the data from off-package DRAM, and 3) inserts the accessed tag and data into the DRAM cache. Therefore, both latency and bandwidth consumption *double* for a miss and a replacement. The original Alloy Cache paper [50] proposes to issue requests to in-package and off-package DRAM in parallel to hide the miss latency. We assume an implementation that serializes the accesses since speculatively accessing off-package DRAM significantly increases the pressure on the already-limited off-package DRAM bandwidth.

Unison Cache [32] stores data at coarse granularity and supports set associativity. The design relies on way prediction to provide low hit latency. On an access, the memory controller reads all of the tags for a set and the data only from the *predicted* way. Loading the data is speculative. On a hit and correct way prediction, the latency is roughly that of a single DRAM access, since the speculative data load is returned together with the tag access. The tag is then stored back to the in-package DRAM with the updated LRU bits. On a miss, latency is doubled due to the extra off-package DRAM access. In terms of bandwidth consumption of a cache miss, Unison Cache 1) loads the tag and the data in the speculative way, 2) performs cache replacement, where it loads a predicted footprint of the page into the in-package DRAM, and 3) stores the updated tags and LRU bits back into the DRAM cache. If the footprint size is large, the bandwidth consumption for a cache miss and a replacement can be dozens of times higher than the that of a cache hit.

2.1.2 Using Address Remapping.

Another technique for tracking data in the DRAM cache is via the virtual-to-physical address mapping mechanism [38, 44] in the page tables. In these designs, data is always managed at page granularity. The physical address space is partitioned between in-package and off-package DRAM. Where a virtual page address maps to can be strictly determined using its physical address. Thus, the system does not perform tag lookups as done in the tag-based designs discussed earlier.

In these page-table-based designs, there are two major challenges. The first challenge is the *TLB coherence.* Whenever a page is inserted into or replaced from the DRAM cache, its virtual-to-physical mapping changes. This change in the mapping must be made coherent across all the TLBs in the system, such that no TLB stores an incorrect, stale mapping. In current systems, this requires a global *TLB shootdown* where each core in the system receives an interrupt and flushes its local TLB. Since a TLB shootdown typically takes multiple microseconds to service [58], frequent TLB shootdowns can severely hurt performance.

The second major challenge in page-table-based designs is what we call *address consistency.* When a virtual page is remapped, its physical address is changed. To avoid having incorrect data in any cache in the system, all the cachelines belonging to the remapped physical page must be removed from all the on-chip SRAM caches. Otherwise, if another virtual page is mapped to the same physical page at a later time, an incorrect value in an SRAM cache may be read by a core, leading to incorrect execution. Removing such cachelines with stale physical addresses on each page remapping can lead to significant performance overhead.

Heterogeneous Memory Architecture (HMA [44]) uses a software-based solution to handle these problems. Periodically, the operating system (OS) ranks all pages and moves hot pages into the in-package DRAM (and cold pages out). The OS updates all page table entries (PTEs), flushes all TLBs for coherence, and flushes cachelines of remapped physical pages from all on-chip caches for address consistency. Due to the high performance overhead of this process, remapping can be done only at a very coarse granularity (e.g., 100 ms to 1 s) to amortize the overhead. Therefore, the DRAM cache replacement policy may not fully capture temporal locality in applications. Also, all programs running in the system have to stop when the pages are moved between in-package and off-package DRAM [44], causing undesirable performance degradation.

Tagless DRAM Cache (TDC [38]) also uses address remapping, but enables frequent cache replacement via a hardware-managed TLB coherence mechanism. Specifically, TDC maintains a TLB directory structure in main memory and updates it whenever an entry is

inserted or removed from *any* TLB in the system. Such fine-grained TLB coherence incurs extra design complexity. Further, the storage cost of the directory may be a potential scalability bottleneck as the core count increases. TDC [38] does not discuss address consistency, so it is unclear which solution, if any, TDC employs for the address consistency problem.

2.2 DRAM Cache Replacement

Cache replacement is another important challenge in in-package DRAM designs. We discuss both hardware and software approaches presented in previous work.

2.2.1 Hardware-Managed. Hardware-managed caches are able to make replacement decisions on each DRAM cache miss, and thus can adapt rapidly to changing workload behavior. Many designs, including Alloy Cache, Unison Cache and TDC, *always* insert the fetched data into the DRAM cache for each cache miss. Although this is common practice for SRAM caches, the incurred extra replacement traffic is expensive for a DRAM cache due to the its limited bandwidth. Some previous designs try to reduce the replacement traffic with a stochastic mechanism [20, 33] where replacement happens with a small probability upon each access. We will use a similar technique in Banshee as well (cf. Section 4.2). For page-granularity DRAM cache designs, frequent replacement also causes *over-fetching*, where a whole page is cached but only a subset of the corresponding cachelines are accessed before eviction. This leads to unnecessary DRAM traffic. The DRAM traffic due to replacement can be even higher than when DRAM caching is completely disabled, leading to large performance degradation, as shown in [33]. To solve this problem, previous works use a sector cache design [40, 52] and rely on a *footprint predictor* [28, 35] to determine which cachelines within a page to load on a cache miss. We show how Banshee improves bandwidth efficiency over these designs in Section 6.

A replacement policy must select a victim to evict. Alloy Cache is direct-mapped, and so has only one victim to replace. Conventional set-associative caches (e.g., Unison Cache) use the Least-Recently-Used (LRU) [32] or a Frequency-Based Replacement (FBR) [33] policy. These policies require additional metadata to track the relative age-of-access or access-frequency for cachelines. Loading and updating the metadata incurs significant DRAM traffic. TDC implements a fully-associative DRAM cache, but uses a FIFO replacement policy, which hurts hit rate for certain applications. Since TDC performs replacement at page granularity for each cache miss, it cannot support large pages efficiently [38].

2.2.2 Software-Managed. Software-based cache replacement algorithms (e.g., HMA [44]) can be relatively sophisticated. Thus, they may perform better than hardware mechanisms at predicting the best data to keep in the DRAM cache. However, they incur significant execution time overhead, and therefore, are generally invoked only periodically (e.g., as in [44]). This makes them much slower to adapt to changing application behavior.

3 BANSHEE DRAM CACHE DESIGN

Banshee aims to maximize bandwidth efficiency for both in-package and off-package DRAM. In this section, we describe how Banshee tracks the contents of the DRAM cache. We describe how Banshee handles cache replacement in Section 4.

3.1 Banshee Architecture

Figure 1 shows the architecture of Banshee. For simplicity, we assume that both the in-package and the off-package DRAM share the same memory controller. Our techniques also work if they have separate memory controllers. The in-package DRAM is a memory-side cache and is not inclusive with respect to the on-chip SRAM caches. Three major changes are made to the hardware and software, and they are highlighted in red. First, PTEs and TLB entries are extended to indicate whether a page is cached, and if so, where it is cached. Second, a new hardware structure, *Tag Buffer*, is added to the memory controller for efficient TLB coherence. Third, the logic in the memory controller is changed for both cache content tracking and cache replacement, as we describe below.

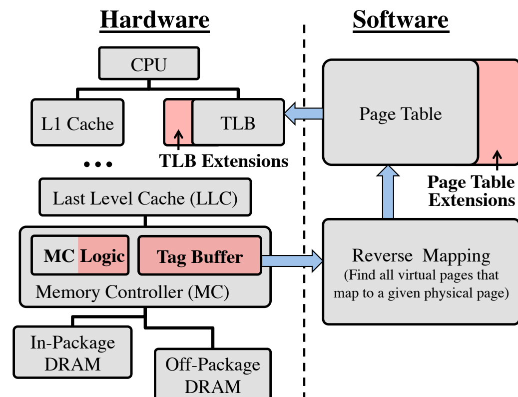


Figure 1: Overall Architecture of Banshee – Changes to hardware and software components are highlighted in red.

Banshee manages the DRAM cache at page granularity and uses the page tables and TLBs to track DRAM cache contents, like TDC [38] and HMA [44]. Unlike previous page-table-based designs, however, Banshee uses the *same* address space for in-package and off-package DRAM to solve the address consistency problem (discussed in Section 2.1.2). Banshee adds extra bits to the corresponding PTE and TLB entry to indicate whether or not a page is cached. This simple change solves the address consistency problem. Because the physical address of a page does *not* change when it is remapped, all cachelines within the page that are present in SRAM caches always have consistent addresses.[4]

To simplify the TLB coherence problem, Banshee implements a *lazy*, software/hardware cooperative TLB coherence protocol using the Tag Buffer. Information of recently-remapped pages is stored only in the Tag Buffer but *not* updated in the corresponding PTEs and TLB entries. All memory requests are able to find the latest mapping information by checking the Tag Buffer at the

[4] In contrast, previous page-table-based DRAM cache designs use *different* physical address spaces for in-package and off-package DRAM. This causes the address consistency problem. If a page is inserted into or evicted from an in-package DRAM, not only do we need to update the PTEs, but all the cachelines in on-chip caches belonging to the affected page need to be updated or invalidated, since their physical addresses have changed.

memory controller. When the Tag Buffer becomes full, the mapping information stored in it propagates to PTEs and TLB entries via software support. Banshee thus significantly reduces the cost of TLB coherence using this mechanism.

3.2 Page Table and TLB Extension

The DRAM cache in Banshee is set-associative. Each PTE is extended with two pieces of mapping information indicating whether and where a page is cached: 1) a new *cached* bit indicates whether a page is resident in the DRAM cache, and 2) some new *way* bits indicate which way the page is cached in.

Every L1 miss carries the mapping information (i.e., the cached and way bits) from the TLB throughout the memory hierarchy. If the request is satisfied before reaching a memory controller, the cached and way bits are simply ignored. If the request misses the LLC and reaches a memory controller, it first looks up the Tag Buffer for the latest mapping of the page. A Tag Buffer miss means the information attached to the request is up-to-date. For a Tag Buffer hit, the mapping information carried with the request is ignored and the correct mapping information from the Tag Buffer is used.

Hardware prefetch requests from the L2 or higher (e.g., L3) levels present a complication. These caches typically operate in the physical address space, and thus cannot access the TLB for the mapping information. In most systems, however, a prefetch of this sort stops at the page boundary [5], since the data beyond that boundary in the physical address space can be unrelated to the data in the adjacent virtual page. Further, these prefetches are usually triggered (directly or indirectly) by demand or prefetch requests coming from the core or the L1 cache. Thus, we can copy the mapping information from a triggering request to the prefetches it triggers.

3.3 Tag Buffer

Banshee adds a Tag Buffer to each memory controller. The Tag Buffer holds the mapping information of recently-remapped pages belonging to that memory controller. Figure 2 shows the organization of a Tag Buffer. It is organized as a small set-associative cache. The physical page number serves as the tag. The *valid* bit indicates whether the entry contains a valid mapping. For a valid entry, the *cached* bit and *way* bits indicate whether and where the page exists in the DRAM cache. The *remap* bit is set for a page whose remapping information is *not* updated in the page table entry for the page. The remap bit enables an optimization we discuss next.

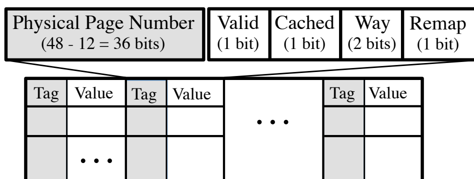


Figure 2: Tag Buffer Organization – The Tag Buffer is organized as a set-associative cache. The DRAM is 4-way set-associative in this example.

Unlike requests arriving at a memory controller, LLC dirty evictions do not carry mapping information. For those, if the mapping information of the evicted cacheline is not in the Tag Buffer, then the memory controller needs to probe the tags stored in the DRAM cache (cf. Section 4.1) to determine if the request is a hit or a miss. These tag probe operations consume DRAM cache bandwidth and may hurt performance.

To reduce such tag probes, we use otherwise-empty entries in the Tag Buffer to hold mappings for pages cached in the LLC. On an LLC miss that also misses in the Tag Buffer, we allocate an entry in the Tag Buffer for the page: the *valid* bit of the entry is set to 1, indicating a useful mapping, but the *remap* bit is set to 0, indicating that the entry stores the *same* mapping as in the PTEs. For a Tag Buffer hit, a dirty LLC eviction does *not* probe the DRAM cache, thereby reducing DRAM traffic. An entry with its remap bit set to 0 can be replaced from the Tag Buffer without affecting correctness (We use the LRU replacement policy among such entries).

3.4 Page Table and TLB Coherence

As the Tag Buffer fills, the mapping information stored in it needs to be propagated to the page table, to make space for future cache replacements. Since the Tag Buffer contains only the physical address of a page, yet the page table is indexed using the virtual address, we need a mechanism to identify all the PTEs corresponding to a physical address.

TDC proposes a hardware-based inverted page table to map a page's physical address to its PTE [38]. This solution, however, cannot handle the *page aliasing* problem where multiple virtual pages are mapped to the same physical page. To identify whether aliasing exists, some internal data structure in the OS (i.e., the page descriptors) has to be accessed, which incurs significant extra overhead.

We observe that a modern operating system *already* has a *reverse mapping* mechanism to quickly identify the associated PTEs for a physical page, regardless of any aliasing. This functionality is necessary to implement page replacement between main memory and secondary storage (e.g., hard disk or solid-state drive) since reclaiming a main memory page frame requires accessing and updating all the PTEs corresponding to it. Reverse mapping in existing systems is implemented via either an inverted page table (e.g., as in Ultra SPARC and PowerPC [57]) or a special reverse mapping mechanism (e.g., Linux [12]). In Banshee, we use this reverse mapping to identify PTEs that map to a given physical page.

When a Tag Buffer fills up to a pre-determined threshold, it sends an interrupt to a randomly-chosen core. The core receiving the interrupt executes a software routine. Specifically, the core reads *all* entries from the Tag Buffers (that have the remap bit set to one) in *all* memory controllers (these entries are memory mapped). The physical address stored in each Tag Buffer entry is used to identify the corresponding PTEs through the reverse mapping mechanism. Then, the *cached* and *way* bits are updated for each PTE, based on their values in the Tag Buffer entry. During this process, the Tag Buffers are locked so that no DRAM cache replacement happens. However, the DRAM can still be accessed and no programs need to stopped (in contrast to prior work [44]).

After all Tag Buffer entries have been propagated to the PTEs, the software routine issues a system wide TLB shootdown to enforce TLB coherence. After this, a message is sent to all Tag Buffers

to clear the *remap* bits for all entries. Note that the mapping information can stay in the Tag Buffer to reduce tag probes for dirty evictions (cf. Section 3.3).

The TLB coherence mechanism described here enforces coherence for the *cached* and *way* bits. These are the only two fields that can be temporarily incoherent and stale. Other fields of a PTE (e.g., physical address, permission bits) are always coherent since Banshee does not change them. Therefore, OS functionalities (e.g., scheduling) that require these PTE fields are not affected when the page tables and TLBs are incoherent.

Depending on a system's software and hardware, the mechanism described above may take tens of thousands of cycles [12]. However, since this cost only needs to be paid when a Tag Buffer is almost full, the cost of TLB coherence is amortized. Furthermore, as we will see in Section 4, remapping pages too often leads to poor performance due to high replacement traffic. Therefore, our design tries to limit the frequency of page remapping, further reducing the occurrence and thus the cost of such TLB coherence.

4 BANDWIDTH-EFFICIENT CACHE REPLACEMENT

As discussed in Section 2.2, the cache replacement policy can significantly affect traffic in both in-package and off-package DRAM. This is especially true for page-granularity DRAM cache designs due to the over-fetching problem within a page. In this section, we propose a new *frequency-based replacement* (FBR) policy that uses *sampling* to achieve a good cache hit rate while minimizing DRAM traffic.

We first discuss the physical layout of the data and metadata in the DRAM cache in Section 4.1. We then describe Banshee's cache replacement algorithm in Section 4.2.

4.1 DRAM Cache Layout

Many previously-proposed tag-based DRAM cache designs store the metadata (e.g., tags, LRU bits) and data in the same DRAM row to exploit row buffer locality, since they always load the metadata along with data. Such an organization can be efficient for a fine-granularity DRAM cache. For a page-granularity DRAM cache, however, pages and tags do not align well within a DRAM row buffer [32], leading to extra design complexity and inefficiency.

In Banshee, the metadata is *not* accessed for each main memory request. Therefore, we store the metadata and the data separately for better alignment. Figure 3 shows the layout of a data row and a metadata row in a DRAM cache that has a row buffer size of 8 KB and a page size of 4 KB. The metadata of each DRAM cache set take 32 bytes in a tag row. For a 4-way associative DRAM cache, each set contains 16 KB of data and 32 bytes of metadata, so the metadata storage overhead is only 0.2%.

Banshee stores metadata for each cached page (the grey portion in Figure 3) and also for a set of *candidate pages* that it is considering to cache. The metadata for each page includes a *tag* and a *count* indicating how frequently the page is accessed. The metadata for a cached page also includes a *valid* and a *dirty* bit. Intuitively, the most frequently accessed pages (i.e., pages with the large frequency counters) should stay in the DRAM cache. Among uncached pages, the ones that are most frequently accessed should be tracked as candidate pages.

4.2 Bandwidth-Aware Replacement Policy

An FBR policy incurs DRAM cache traffic by 1) reading and updating the frequency counters and 2) replacing data. In Section 4.2.1, we introduce a sampling-based counter maintenance scheme to reduce the counter read/update traffic. In Section 4.2.2, we discuss our new bandwidth-aware replacement algorithm that aims to minimize replacement traffic while maintaining a good cache hit rate.

4.2.1 Sampling-Based Counter Updates. In a standard FBR policy [36, 51], each access to a page increments the associated page's frequency counter. We observe that incrementing the counter for each access is *not* necessary. Instead, an access in Banshee updates the frequency counter only with a certain *sample rate*. For a *sample rate* of 10%, for example, the frequency counters are accessed/updated once for every 10 main memory accesses. This reduces counter read/update traffic by 10×. Furthermore, since sampling slows down the incrementing rate of the counters, we can use fewer bits to represent each counter.

It may seem that updating the counters via such sampling leads to inaccurate detection of "hot" pages. However, the vast majority of applications exhibit some spatial locality. When a request to a 64-byte L1 cacheline misses in the DRAM cache, other cachelines belonging to the same page (there are 64 of them for a 4 KB page) are likely to be accessed soon as well. Each of these accesses has a chance to update the frequency counter of the page. In fact, without sampling, we find that counters quickly reach large values and only the high order bits matter for replacement decisions. Sampling effectively "discards" the need to store and track the low-order bits of each counter, which have little useful information anyway.

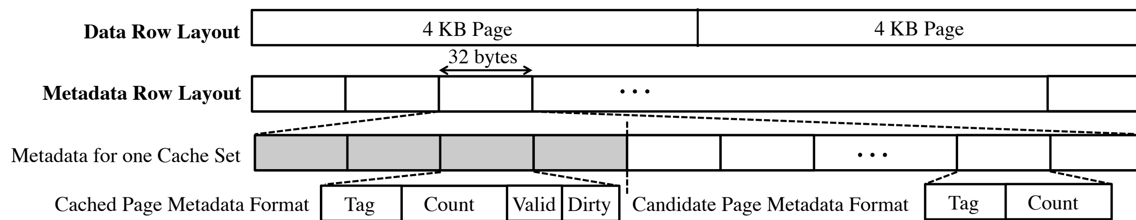


Figure 3: Banshee DRAM Cache Layout (Not Drawn to Scale) – An example 4-way set-associative DRAM cache with an 8 KB row buffer and 4 KB pages. Data and metadata are stored separately on different rows. Metadata is also kept for some pages that are currently *not* cached in the DRAM cache but are considered to be cached, called *candidate* pages.

We further observe that when the DRAM cache works well, i.e., it has a low miss rate, replacement should be rare and the counters need not be frequently updated. Therefore, Banshee uses an adaptive sample rate which is the product of the cache miss rate (tracked dynamically) and a constant *sampling coefficient*.

4.2.2 Replacement Algorithm. DRAM cache replacement can be expensive, in terms of memory traffic, for coarse-granularity designs. For each replacement, the memory controller transfers an entire page from off-package DRAM to in-package DRAM. If a page has poor spatial locality (i.e., the page experiences only a few accesses before being replaced), the in-package and off-package DRAM traffic due to replacement can be even higher than when the DRAM cache is not present. This leads to performance degradation (cf. Section 6).

Frequency-based replacement does not inherently solve this problem because the algorithm may repeatedly keep replacing the least-frequently-accessed page in the cache with a candidate that has a larger frequency counter. When pages have similar counter values, a large number of such replacements can be triggered, thrashing the cache and wasting valuable in-package and off-package DRAM bandwidth.

Banshee solves this problem by replacing a page only when the incoming page's counter is greater than the potential victim page's counter *by a certain threshold*. This ensures that a page just evicted from the DRAM cache is expected to be accessed for ($2 \times threshold/sampling\ rate$) times before it can enter the cache again (i.e., the page's frequency counter becomes greater than a cached page's frequency counter by *threshold*). This prevents a page from entering and leaving the cache frequently. Note that reducing the frequency of replacement also increases the time between Tag Buffer overflows, indirectly reducing the overhead of TLB coherence.

Algorithm 1 shows the complete cache replacement algorithm of Banshee. For each request coming from the LLC, a random number is generated to determine whether the current access should be sampled (line 3). If it is not sampled, which is the common case, then the access is made to the proper DRAM (in-package or off-package) directly. No metadata is accessed and no replacement happens.

If the current access is sampled, then the metadata for the corresponding set is loaded from the DRAM cache to the memory controller (line 4). If the currently-accessed page exists in the metadata (line 5), its counter is incremented (line 6). If the current page is one of the candidate pages and its counter is greater than a cached page's counter by a threshold, then cache replacement should happen (lines 7-9). By default, the threshold is the product of the number of cachelines in a page and the sampling coefficient divided by two ($threshold = page_size \times sampling_coeff\ /\ 2$). Intuitively, this means that replacement can happen only if the benefit of swapping the pages outweighs the cost of the replacement operation. If a counter saturates after being incremented, all counters in the metadata are halved by a shift operation in hardware (lines 10-15).

If the current page does *not* exist in the metadata (line 17), then one of the candidate pages is randomly selected as the victim (line 19). However, Banshee does *not* always replace the chosen victim: instead, the current page can only replace the chosen victim with a certain probability, which decreases as the victim's counter gets larger (line 20-24). This way, it is less likely that a *warm* candidate page is evicted.

Algorithm 1: Banshee Cache Replacement Algorithm

```
1  Input : tag
2  # rand(): random number between 0 and 1.0
3  if rand() < cache_miss_rate × sampling_coefficient then
4      metadata = dram_cache.loadMetadata(tag)
5      if tag is in metadata then
6          metadata[tag].count ++
7          if tag is in metadata.candidates and metadata[tag].count >
             metadata.cached.minCount() + threshold then
8              Insert the page being requested into the DRAM cache. Evict the
                 page with the minimum count from the DRAM cache. Swap the
                 metadata of the two pages.
9          end
10         if metadata[tag].count == max_count_value then
11             # Counter overflow, divide by 2
12             forall t in metadata.tags do
13                 metadata[t].count /= 2
14             end
15         end
16         dram_cache.storeMetadata(tag, metadata)
17     else
18         # The page is not tracked in the tag
19         victim = pick a random page in metadata.candidates
20         if rand() < (1 / victim.count) then
21             Replace victim with the page being requested.
22         end
23     end
24 end
```

5 BANSHEE EXTENSIONS

In this section, we discuss two important extensions of Banshee to support large pages and multiple sockets.

5.1 Supporting Large Pages

Large pages have been widely used to reduce TLB misses and therefore should be supported in DRAM caches. For designs that use tags to track the DRAM cache contents (e.g., Unison Cache and Alloy Cache), supporting large pages does not require changes to the hardware. A large page is simply broken down into and managed as smaller pages or cachelines. For page-table-based designs, however, a large page should be cached either in its entirety or not at all, since its mapping is stored in a single page table entry. In this section, we describe how Banshee supports large pages. The mechanism works for any page size.

In Banshee, the DRAM cache can be partitioned for different-sized pages. Partitioning can happen at context switch time by the OS, which knows how many large pages each process is using. Partitioning can also be done dynamically using runtime statistics based on access counts and hit rates for different page sizes. Since most of our applications either make very heavy use of large pages, or very light use, partitioning could give either most or almost none of the cache, respectively, to large pages. We leave a thorough exploration of Banshee cache partitioning policies for future work.

We force each page (regular or large) to map to a single MC (memory controller) to simplify the management of frequency counters and cache replacement. Bits are attached to each cacheline in the LLC to indicate the size of the page it belongs to. A processor request or an LLC dirty eviction uses the page size information to

determine which MC to access (e.g., by hashing the page number of the accessed page). When the OS reconfigures the large pages, which happens very rarely [11], all lines within the reconfigured pages should be flushed from the on-chip caches and the in-package DRAM cache.

In terms of the DRAM cache layout, a large page mapped to a particular way spans multiple cache sets occupying the corresponding way in each set. One difference between regular and large pages is the cache replacement policy. Banshee manages large page replacement just like regular page replacement using frequency counters with sampling. However, since replacing a large page consumes more memory bandwidth than replacing a regular page, we use a greater threshold when comparing frequency counters of large pages. We also reduce the sample rate of updating frequency counters to prevent counter overflow. Note that large pages do *not* work well for page-table-based schemes that perform replacement on *each* DRAM cache miss. TDC, for example, *disables* caching of large pages for this reason.

5.2 Multi-Socket Support

In a multi-socket shared-memory system, each socket has its own DRAM cache. We can design the DRAM caches to be either *coherent* (i.e., the DRAM cache at one socket can cache pages resident in another socket's off-package DRAM) or *partitioned* (i.e., a DRAM cache caches only the data resident in the *local* socket's off-package DRAM). We discuss both designs here.

Coherent DRAM caches. If the DRAM caches are coherent, then we need a separate DRAM cache coherence protocol [21, 26] for DRAM caches in different sockets. We use such a protocol in our multi-socket design. In addition, with a page-table-based DRAM cache design, like Banshee, the page tables and TLBs need to be kept coherent *across sockets* as well. To indicate whether a page is stored in *any* socket's local DRAM cache, the mapping information in each PTE and each Tag Buffer entry is extended to contain the *cached* and *way* bits for *all sockets* in the system. The lazy TLB coherence mechanism in Banshee (Section 3.4) can be easily enhanced to simplify TLB coherence in the multi-socket scenario, as follows. When a page's mapping information changes (e.g., the page is inserted into or removed from a DRAM cache), the updated mapping information is temporarily stored in the Tag Buffer. This information is lazily propagated to the PTEs and TLBs of other cores/sockets, i,e., only when one of the Tag Buffers in the entire multi-socket system is full. This simple extension of our lazy TLB coherence protocol greatly reduces the overhead of TLB coherence in a multi-socket system. We leave its detailed evaluation for future work.

Partitioned DRAM Caches. Keeping DRAM caches, Page Tables, and TLBs coherent across multiple sockets allows the DRAM caches to be shared across sockets. However, such a solution requires a complex DRAM cache coherence protocol. When the number of sockets is large, it also incurs significant storage overhead because an entry in a TLB or a page table includes the mapping information for *all* sockets. The *partitioned* DRAM cache design is simpler because it restricts the DRAM cache to store only data resident in the *local* socket's off-package DRAM. This eliminates the complexity of handling DRAM cache coherence since data is never replicated in in-package DRAM in different sockets. With good NUMA allocation policies and properly-tuned applications, the vast majority of accesses are to local DRAM, in which case the partitioned DRAM cache design is likely good enough to exploit most of the benefits of DRAM caching.

6 EVALUATION

We evaluate the performance of Banshee and compare it to three state-of-the-art DRAM cache designs. Section 6.1 discusses the methodology of our experiments. Sections 6.2 and 6.3 show the performance and DRAM bandwidth consumption of different DRAM cache designs. Section 6.4 shows the effect of our extensions to Banshee. Section 6.5 presents sensitivity studies.

6.1 Methodology

We use ZSim [53] to simulate a multi-core processor whose configuration is shown in Table 2.[5] The chip has one channel of off-package DRAM and four channels of in-package DRAM. We assume that the timing and bandwidth characteristics of all channels are the same, to model the behavior of in-package DRAM [29, 46, 56]. The maximum bandwidth this configuration offers is 21 GB/s for off-package DRAM and 84 GB/s for in-package DRAM. In comparison, Intel's Knights Landing [55] has roughly 4× the bandwidth and the number of cores (72 cores, 90 GB/s off-package DRAM and 300+ GB/s in-package DRAM bandwidth), so our baseline configuration has roughly the same bandwidth per core.

Table 2: System Configuration.

System Configuration	
Core Frequency	2.7 GHz
Number of Cores	16
Core Model	4-issue, out-of-order
Memory Subsystem	
Cacheline Size	64 bytes
L1 I Cache	32 KB, 4-way
L1 D Cache	32 KB, 8-way
L2 Cache	128 KB, 8-way
Shared L3 Cache	8 MB, 16-way
DRAM	
Off-Package DRAM	1 channel
In-Package DRAM	4 channels, 256 MB per channel
Rank	4 ranks per channel
Bank	8 banks per rank
Bus Frequency	667 MHz (DDR 1333 MHz)
Bus Width	128 bits per channel
tCAS-tRCD-tRP-tRAS	10-10-10-24

Table 3 shows the default parameters of Banshee. The DRAM cache is 4-way set-associative. Each PTE and TLB entry is extended with 3 bits (i.e., one *cached* bit and two *way* bits) for the mapping information. A fully-associative TLB entry needs to at least store the virtual and physical page number for the corresponding page, which require 64 - 12 = 52 bits and 48 - 12 = 36 bits respectively (assuming 64-bit virtual address space and 48-bit physical address space with 4 KB page size). Therefore, the storage overhead of the TLB extension is only 3.4%. The storage overhead of the PTE extension is *zero* since we are using otherwise-unused bits. Each request in the memory hierarchy carries the 3 mapping bits. Each

[5] The source code of our simulator and the implementation of evaluated DRAM cache designs are available at https://github.com/yxymit/banshee

memory controller has an 8-way set associative Tag Buffer with 1024 entries, requiring only 5 KB storage per memory controller. The memory controller triggers a "Tag Buffer full" interrupt when the buffer is 70% full. We assume the interrupt handler runs on a single randomly-chosen core and takes 20 microseconds. For a TLB shootdown, the initiating core (i.e., initiator) is unavailable for program execution for 4 microseconds and every other core (i.e., slave) is unavailable for 1 microsecond [58].

Table 3: Banshee Configuration.

DRAM Cache and Tags	
Associativity	4
Page Size	4 KB
Tag Buffer	1 Tag Buffer per MC 8-way, 1024 entries Flushed when 70% full
PTE Coherence Overhead	20 μs
TLB Shootdown Overhead	Initiator 4 μs, slave 1 μs
Cache Replacement Policy	
Metadata per DRAM Cache Set	Tags and frequency counters for 4 cached pages and 5 candidate pages
Frequency Counter	5 bits
Sampling Coefficient	10%

Each frequency counter is 5-bit wide. The 32-byte per-set metadata holds information for 4 cached pages and 5 candidate pages.[6] The default sampling coefficient is 10% – the actual sample rate is this multiplied by the DRAM cache miss rate observed for the last one million memory accesses.

6.1.1 Baselines.

We compare Banshee to five baselines.

No Cache: The system contains only off-package DRAM.

Cache Only: The system contains only in-package DRAM with infinite capacity.

Alloy Cache [50]: A fine-granularity design, described in Section 2. We also include the *bandwidth-efficient cache fills* and the *bandwidth-efficient writeback probe* optimizations from BEAR [20] to improve bandwidth efficiency. This includes a stochastic replacement mechanism that performs replacement with only a 10% probability. In some experiments, we show results for always replacing (`Alloy 1`), and replacing only 10% of the time (`Alloy 0.1`).

Unison Cache [32]: A state-of-the-art coarse-granularity design, described in Section 2.1.1. We model an LRU replacement policy. We implement an oracular footprint prediction and assume *perfect* way prediction. For footprint prediction, we first profile each workload offline to collect the average number of blocks touched per page fill, as the footprint size; in the actual experiment, each replacement moves *only* the number of cachelines specified by the footprint size. The footprint is managed at a 4-line granularity. We assume the predictors incur no latency or bandwidth overhead.

Tagless DRAM Cache (TDC) [38]: A state-of-the-art page-granularity design, described in Section 2.1.2. We model an idealized TDC configuration. Specifically, we assume a zero-performance-overhead TLB coherence mechanism and ignore all the side effects of the mechanism (i.e., address consistency, page aliasing). We also implement the same oracular footprint predictor for TDC just as we do for Unison Cache.

[6] With a 48-bit address space and the DRAM cache parameters, the tag size is 48 - 16 (2^{16} sets) - 12 (page offset) = 20 bits. Each cached page 20 + 5 + 1 + 1 = 27 bits of metadata and each candidate page has 25 bits of metadata (Figure 3).

6.1.2 Benchmarks.

We use SPEC CPU2006 [25] and graph analytics benchmarks [60].

We select a subset of SPEC benchmarks that have large memory footprints. We consider both homogeneous and heterogeneous multi-programmed workloads. For homogeneous workloads, each core in the simulated system executes one instance of a benchmark and all the instances run in parallel. Heterogeneous workloads model a multi-programmed environment where the cores run a mix of benchmarks. We use three randomly-selected mixes, shown in Table 4.

Table 4: Mixed SPEC Workloads.

Name	Benchmarks in mix
Mix1	libq-mcf-soplex-milc-bwaves-lbm-omnetpp-gcc × 2
Mix2	libq-mcf-soplex-milc-lbm-omnetpp-gems-bzip2 × 2
Mix3	mcf-soplex-milc-bwaves-gcc-lbm-leslie-cactus × 2

To represent throughput computing workloads, the target applications for modern systems employing in-package DRAM [2, 56], we evaluate multi-threaded graph analytics workloads. We use all graph workloads from [60].

In our experiments, Each graph benchmark runs to completion and each combination of SPEC benchmarks runs for 100 billion instructions across all the cores. We warm up the DRAM cache until it is full. By default, all benchmarks use 4 KB pages only.

Many benchmarks that we evaluate have very high memory bandwidth requirements. With the `CacheOnly` configuration, for example, 10 of the 16 benchmarks have an average DRAM bandwidth consumption of over 50 GB/s (bursts may exceed this). This bandwidth requirement exerts enough pressure on the in-package DRAM (with a maximum bandwidth of 85 GB/s) such that main memory requests experience high latency due to contention. Our memory-intensive benchmarks (e.g., pagerank, lbm, libquantum) experience 2–4× higher memory access latency compared to compute-intensive benchmarks (e.g., gcc, sgd, soplex) due to the memory bandwidth bottleneck.

6.2 Performance

Figure 4 shows the speedup of different cache designs normalized to `NoCache`. The geo-mean bars indicate geometric mean across all the workloads. On average, Banshee provides a 68.9% speedup over Unison Cache, 26.1% over TDC and 15.0% over Alloy Cache. Improved bandwidth efficiency is the main contributor to the performance improvement (as we will show in Section 6.3). Compared to Unison Cache and Alloy Cache, Banshee also reduces the cache miss latency since the DRAM cache is not probed to check tags for a cache miss.

Unison Cache and TDC have worse performance than other designs on some benchmarks (e.g., `omnetpp` and `milc`). These benchmarks have poor spatial locality. As Unison Cache and TDC replace a whole page of data for each DRAM cache miss, they generate unnecessary DRAM traffic for cache replacement. Having a footprint predictor helps but does not completely solve the problem since the footprint *cannot* be managed at fine granularity due to the storage overhead (we model a 4-line granularity). Banshee also

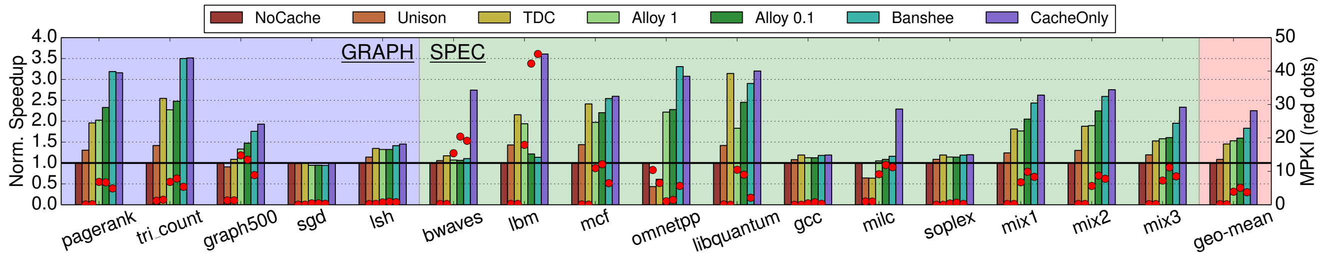


Figure 4: Speedup Normalized to NoCache – Speedup is shown in bars and misses per kilo instruction (MPKI) is shown in red dots.

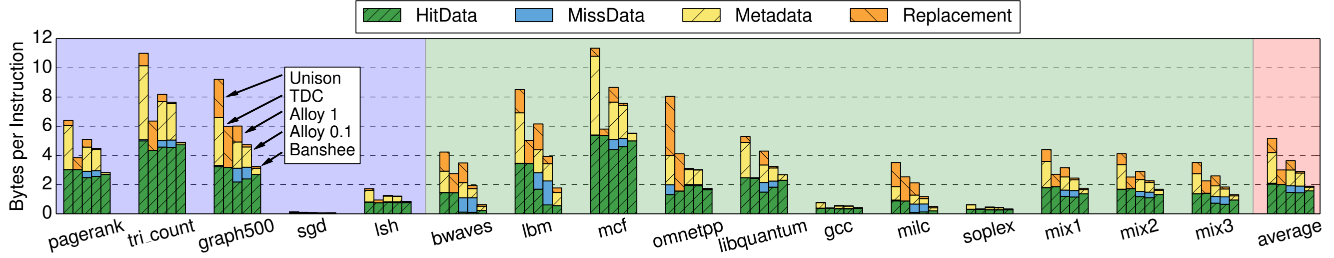


Figure 5: In-package DRAM Traffic Breakdown.

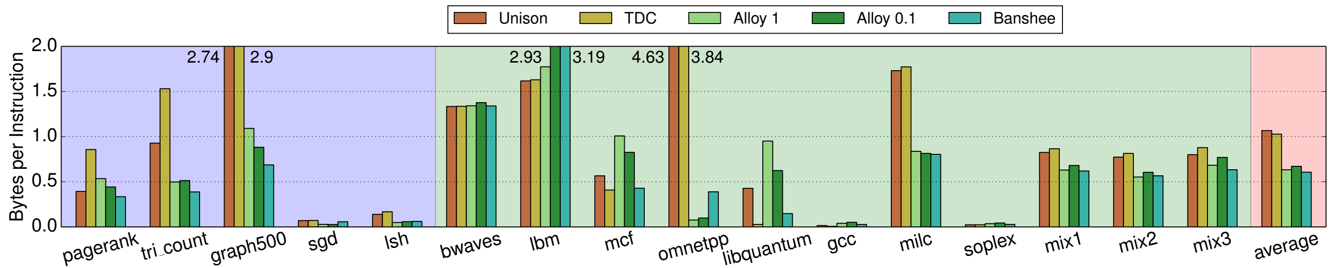


Figure 6: Off-package DRAM Traffic.

uses page granularity, but its bandwidth-aware replacement policy significantly reduces unnecessary replacement traffic for these benchmarks.

On `lbm`, both Banshee and `Alloy 0.1` give worse performance than other baselines. `lbm` has very good spatial locality on each page, but a page is accessed only a small number of times before it gets evicted. `Alloy 1`, Unison Cache and TDC have good performance on `lbm` since they perform replacement for *every* DRAM cache miss, thereby exploiting the existing locality well. Banshee and `Alloy 0.1`, in contrast, cannot exploit all the locality due to their *selective* caching mechanisms. One solution is to dynamically switch between different replacement policies based on a program's access pattern. For example, we can design some pre-determined sets in the cache to use different replacement policies and select the best-performing policy for the rest of the cache, called set sampling/dueling [20, 48, 49]. We leave the exploration of such ideas for future work.

The red dots in Figure 4 show the DRAM cache *Misses Per Kilo Instruction* (MPKI) for each DRAM cache scheme (except `NoCache` and `CacheOnly`). Unison Cache and TDC have very low miss rates since they can exploit spatial locality within a page. However, their high memory bandwidth consumption due to frequent DRAM cache replacement offsets the benefits of low miss rates (cf. Section 6.3). Alloy Cache and Banshee have higher miss rates. The miss rate is high for Alloy Cache because it is managed at fine granularity, and therefore it cannot exploit spatial locality well. The miss rate is high for Banshee because of its bandwidth-efficient cache replacement policy, which does not perform cache replacement for every DRAM cache miss.

For some benchmarks (e.g., pagerank, omnetpp), Banshee performs even better than `CacheOnly`. This is because `CacheOnly` has no external DRAM. Thus, `CacheOnly`'s total available DRAM bandwidth is lower than Banshee's which has both in-package and off-package DRAM. We provide more discussion on balancing DRAM bandwidth in Section 6.4.2.

6.3 DRAM Traffic

Figures 5 and 6 show, respectively, the in-package and off-package DRAM traffic. Traffic is measured in *bytes per instruction* to convey the memory intensity of a workload, in addition to comparing the bandwidth efficiency of different cache designs.

In Figure 5, `HitData` is the DRAM cache traffic for DRAM cache hits. This is the only useful data transfer; everything else can be considered as overhead. `Metadata` is the traffic for metadata accesses (e.g., tags, LRU bits, frequency counters). `Replacement` is the traffic for DRAM cache replacement. For Alloy and Unison Cache, `MissData` is the traffic to load data from the DRAM cache

when a cache miss happens. MissData exists because, in both designs, a request to a memory controller loads both the tag and the data from the DRAM cache. The tag is loaded to check whether the request is a hit or a miss; the data is loaded speculatively to hide latency if it is a DRAM cache hit (cf. Section 2.1.1). For a DRAM cache miss, however, speculatively loading the data (i.e., MissData) consumes DRAM cache bandwidth unnecessarily.

Both Unison and Alloy Cache incur significant traffic for tag accesses. Alloy Cache also consumes considerable traffic for MissData at cache misses. Unison Cache has small MissData traffic due to its low miss rate. Both schemes also require significant replacement traffic. Stochastic replacement (Alloy 0.1) reduces Alloy Cache's replacement traffic, but other overheads still remain.

TDC eliminates the tag traffic by managing mapping information using page tables and TLBs. However, like Unison Cache, it still incurs significant traffic for DRAM cache replacement. For most benchmarks, the traffic difference between Unison and TDC is mainly the removal of Metadata traffic. On some benchmarks (e.g., mcf, libquantum), TDC incurs less replacement traffic than Unison Cache because of its higher hit rate due to full associativity. On some other benchmarks (e.g., pagerank, tri_count), however, TDC incurs more traffic due to FIFO replacement. Overall, we find that the replacement traffic reduces the performance of both Unison Cache and TDC.

Because of our bandwidth-aware replacement policy, Banshee provides significantly better bandwidth efficiency for in-package DRAM (35.8% less traffic than the best baseline, i.e., Alloy 0.1). Banshee achieves this without incurring extra off-package DRAM traffic (shown in Figure 6), which is a major reason why Banshee provides the best performance. On average, Banshee's off-package DRAM traffic is 9.7% lower than the best previous scheme in terms of performance (Alloy 0.1), 3.1% lower than the best previous scheme in terms of off-package DRAM traffic (Alloy 1), 42.4% lower than Unison Cache, and 43.2% lower than TDC.

Graph Processing Workloads. As mentioned earlier, graph processing workloads are arguably more important for our modeled system, which is targeted towards throughput computing workloads. We observe that for graph codes with high main memory traffic (i.e., pagerank, tri_count and graph500), Banshee provides some of its largest performance improvements over the best baseline, while also significantly reducing both in-package and off-package DRAM traffic compared to all baseline schemes.

We conclude that Banshee is very effective at improving both system performance and main memory bandwidth efficiency.

6.4 Banshee Extensions

6.4.1 Supporting Large Pages. Banshee can efficiently support large pages, as discussed in Section 5.1. Here, we evaluate the performance impact of large pages on Banshee. To simplify the evaluation, we assume that all data resides in large (2 MB) pages. The sampling coefficient is chosen to be 0.001 and the replacement threshold is calculated accordingly (Section 4.2.2). When comparing performance with large and small pages, we assume perfect TLBs to isolate the impact of the DRAM subsystem.

Our evaluation shows that on graph analytics benchmarks, with large pages, Banshee's performance is on average 3.6% higher than the baseline Banshee with 4 KB pages. The performance gain of Banshee with large pages comes from 1) the more accurate hot page detection at the larger page granularity, 2) the reduced frequency counter updates, and 3) the reduced TLB coherence overhead.

6.4.2 Balancing DRAM Bandwidth. Some related works [7, 8, 19] propose to balance the accesses to in-package and off-package DRAM in order to maximize the overall bandwidth efficiency. These optimizations are orthogonal to Banshee and can be used in combination with Banshee.

We implemented the technique from BATMAN [19], which turns off parts of the in-package DRAM if it has too much traffic (i.e., over 80% of the total DRAM traffic). On average, this optimization leads to 5% (up to 24%) performance improvement for Alloy Cache and 1% (up to 11%) performance improvement for Banshee. The gain is smaller in Banshee since it has less total bandwidth consumption to begin with. Even with bandwidth balancing used in both designs, Banshee still outperforms Alloy Cache by 12.4%.

6.5 Sensitivity Studies

In this section, we study the performance of Banshee with different design parameters.

6.5.1 DRAM Cache Replacement Policy. Figure 7 shows the normalized performance and in-package DRAM traffic of different replacement policies to provide insight into where the performance gain of Banshee is coming from.

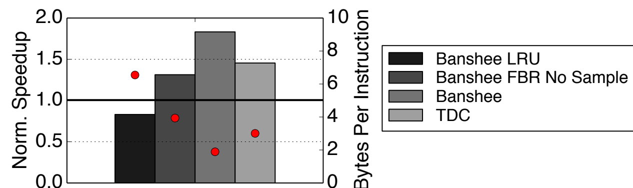


Figure 7: Sensitivity of Banshee to Cache Replacement Policy – Speedup normalized to NoCache (bars) and in-package DRAM traffic (red dots) of different replacement policies on Banshee. Results averaged over all benchmarks.

Banshee LRU uses an LRU policy similar to Unison Cache but does not use a footprint cache. It has low performance and high bandwidth consumption due to frequent page replacements which occur on every miss.

Using frequency-based replacement improves performance and bandwidth efficiency over LRU since only hot pages are cached. However, if the frequency counters are updated on every DRAM cache access (Banshee FBR No Sample, similar to CHOP [33]), significant metadata traffic is incurred, which leads to performance degradation. We conclude that both FBR and sampling-based counter management should be used to achieve good performance in Banshee.

6.5.2 Page Table Update Overhead. One potential disadvantage of Banshee is the overhead of updating page tables when enforcing TLB coherence (cf. Section 3.4). However, this cost is paid only when the Tag Buffer fills up after many page remappings. Furthermore, our replacement policy intentionally slows remapping

(cf. Section 4). On average, the page table update is triggered once every 14 milliseconds, which has low overhead in practice.

Table 5 shows the average and maximum performance degradation across our benchmarks, compared to an ideal baseline that incurs zero cost for page table updates, for a range of page table update costs. The average performance degradation is less than 1%, and scales sublinearly with the page table update cost. We find that doubling the Tag Buffer size has a similar effect on performance as reducing the page table update cost by half (not shown).

Table 5: Sensitivity to Page Table Update Cost.

Update Cost (μs)	Avg Perf. Loss	Max Perf. Loss
10	0.11%	0.76%
20	0.18%	1.3%
40	0.31%	2.4%

6.5.3 *Storing Tags in SRAM.*

For systems with small DRAM caches, storing tags in on-chip SRAM may also be a good design option. Compared to the page-table-based mapping management in Banshee, storing tags in SRAM incurs higher latency and more storage requirement, but doing so reduces the design complexity. Note that the bandwidth-aware FBR policy proposed in this paper is orthogonal to mapping management. Therefore, Banshee's replacement policy can improve DRAM bandwidth efficiency for designs that store tags in SRAM.

We evaluated a version of Banshee where the tags are stored in SRAM instead of page tables and TLBs. For a 1 GB DRAM cache, the tags and FBR metadata consume 2 MB of SRAM storage (cf. Figure 3) which is 1/4th of the LLC size. We assume the tag array lookup latency to be the same as the LLC latency. With SRAM tags and bandwidth-efficient FBR, the performance of this version of Banshee is on average 3% (up to 10%) worse than our proposed Banshee design. Banshee with SRAM tags still outperforms other baseline DRAM cache designs.

6.5.4 *DRAM Cache Latency and Bandwidth.*

Figure 8 shows the performance (normalized to `NoCache`) of different DRAM cache schemes for different DRAM cache latency and bandwidth parameters. Each data point is the geometric mean performance over all benchmarks. The x-axis of each figure shows the latency and bandwidth of in-package DRAM relative to off-package DRAM. By default, we assume in-package DRAM has the same latency as and 4× the bandwidth of off-package DRAM.

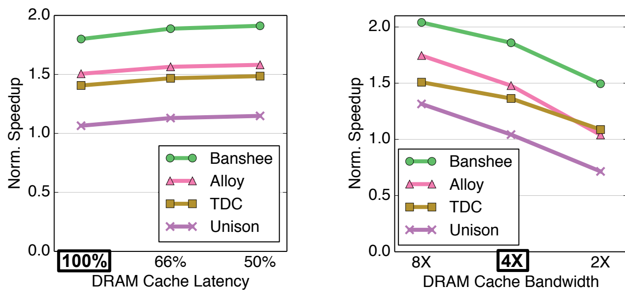
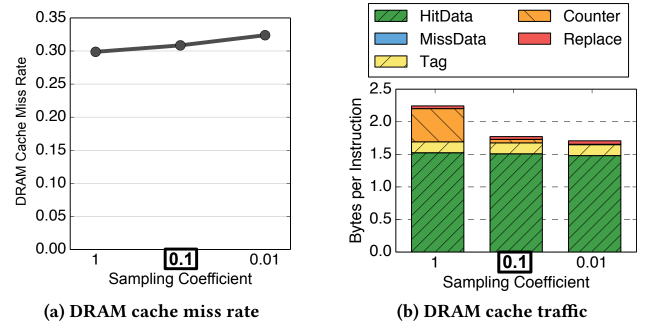


Figure 8: Sensitivity to DRAM Cache Latency and Bandwidth – Each data point is the geometric mean over all benchmarks. Default parameter setting is highlighted on x-axis. Latency and bandwidth values are relative to off-package DRAM.

As the in-package DRAM's latency decreases and bandwidth increases, the performance of all DRAM cache schemes improves. We observe that performance is more sensitive to bandwidth than to zero-load latency. Although not shown in the figure, changing the core count in the system has a similar effect as changing the DRAM cache bandwidth. Since Banshee's performance gain over the baselines is more significant when the DRAM cache bandwidth is more limited, we expect Banshee to have larger performance gain with more cores.

6.5.5 *Sampling Coefficient.*

Figure 9 shows the DRAM cache miss rate (fraction of accesses that miss) and traffic breakdown for different values of the *sampling coefficient* in Banshee. As the sampling coefficient decreases, miss rate increases and the amount of traffic for updating the frequency counters (`Counter`) decreases. We chose 0.1 as the default sampling coefficient since it has reasonably low miss rate, and the incurred traffic overhead is small enough.



(a) DRAM cache miss rate **(b) DRAM cache traffic**

Figure 9: Sensitivity of Banshee to Sampling Coefficient – The default sampling coefficient is 0.1.

6.5.6 *Associativity.*

Table 6 shows the cache miss rate for different values of set-associativity in Banshee. Doubling the number of ways requires adding one more bit to each PTE, and doubles the per-set metadata. Higher associativity reduces the cache miss rate. Since we observe diminishing miss rate reduction with more than four ways, we choose the 4-way set-associative DRAM cache as our default design point.

Table 6: *Cache Miss Rate vs. Associativity in Banshee*

Associativity	1 way	2 ways	4 ways	8 ways
Miss Rate	36.1%	32.5%	30.9%	30.7%

7 RELATED WORK

Besides those discussed in detail in Section 2, other DRAM cache designs are proposed in the literature. PoM [54] and CAMEO [18] manage in-package and off-package DRAM in different address spaces at fine (64 B) granularity. Tag Tables [23] compress the tag storage for Alloy Cache to make it cheaper to put in on-chip SRAM. Bi-Modal Cache [24] supports heterogeneous block sizes (cacheline and page) to get the best of both worlds. All these schemes focus on minimizing latency of the design and incur significant traffic for tag lookups and/or cache replacement.

Similar to this paper, several other papers propose DRAM cache designs with optimizations to improve bandwidth efficiency. CHOP [33] targets the off-package DRAM bandwidth bottleneck

for page-granularity DRAM caches, and uses FBR instead of LRU. However, their scheme still incurs significant traffic for counter updates (cf. Section 6.5.1), whereas Banshee uses sampling-based counter management and bandwidth-aware replacement to reduce such traffic. Several other papers propose to improve off-package DRAM traffic for page-granularity DRAM caches using a *footprint cache* [28, 31, 32]. As we showed in Section 6, however, a footprint cache alone cannot eliminate all unnecessary replacement traffic. That said, the footprint cache idea is orthogonal to Banshee and can be combined with Banshee for even better performance. A few papers [7, 8, 19] propose to balance the bandwidth utilization between in-package and off-package DRAM to maximize efficiency. As we evaluated in Section 6.4.2, these techniques are orthogonal to Banshee.

BEAR [20] improves Alloy Cache's DRAM cache bandwidth efficiency. Our implementation of Alloy Cache already includes some of the key BEAR optimizations. These optimizations cannot eliminate all tag lookups, and, as we have shown in Section 6.3, Banshee provides higher DRAM cache bandwidth efficiency as well as higher performance.

Several other works consider heterogeneous memory technologies beyond in-package DRAM. These include designs for hybrid DRAM and Phase Change Memory (PCM) [22, 39, 45, 59], a single DRAM chip with fast and slow portions [13–15, 37, 42], and the design of different off-chip DRAM channels with various different characteristics (e.g., latency, reliability, power consumption) [16, 43, 47] .We believe the ideas in this paper can be applied to such heterogeneous memory systems, as well.

Among all previous designs, TDC [38] is the one closest to Banshee. Both TDC and Banshee use page tables and TLBs to track data mapping at page granularity. The key novelty of Banshee compared to TDC is 1) the bandwidth-efficient frequency-based DRAM cache replacement policy, 2) the low-overhead lazy TLB coherence mechanism using the Tag Buffer, and 3) using the *same* address space for in-package and off-package DRAM to solve the address consistency problem. As a result, Banshee significantly reduces the large TLB coherence overheads and the unnecessary bandwidth consumption in both in-package and off-package DRAM, thereby boosting system performance over a wide range of workloads.

8 CONCLUSION

We propose a new DRAM cache design called Banshee. Banshee aims to maximize both in-package and off-package DRAM bandwidth efficiency and performs better than previous latency-optimized DRAM cache designs on memory-bound applications. Banshee achieves this through a software/hardware co-design approach. Specifically, Banshee uses a new, low-overhead *lazy TLB coherence* mechanism and *a bandwidth-aware DRAM cache replacement* policy to minimize the memory bandwidth overhead for 1) tracking the DRAM cache contents, and 2) performing DRAM cache replacement. Our extensive experimental results show that Banshee provides significant performance and bandwidth efficiency improvements over three state-of-the-art DRAM cache schemes.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers of MICRO 2017, ISCA 2017, HPCA 2017 and MICRO 2016 for their helpful feedback. An earlier version of this work was posted on arXiv in 2017 [61]. This work is supported in part by the Intel Science and Technology Center (ISTC) for Big Data.

REFERENCES

[1] Hybrid Memory Cube Specification 2.1. http://www.hybridmemorycube.org, 2014.

[2] NVLink, Pascal and Stacked Memory: Feeding the Appetite for Big Data. https://goo.gl/y6oYqD, 2014.

[3] The Road to the AMD "Fiji" GPU. https://goo.gl/ci9BvG, 2015.

[4] Data Sheet: Tesla P100. https://goo.gl/Y6gfXZ, 2016.

[5] Intel®64 and IA-32 Architectures Optimization Reference Manual. https://goo.gl/WKkFiw, 2016.

[6] NVidia Tesla V100 GPU Accelerator. https://goo.gl/5eqTg5, 2017.

[7] Agarwal, N., et al. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *ASPLOS* (2015).

[8] Agarwal, N., et al. Unlocking Bandwidth for GPUs in CC-NUMA Systems. In *HPCA* (2015).

[9] Ahn, J., et al. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *ISCA* (2015).

[10] Ahn, J., et al. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *ISCA* (2015).

[11] Bailey, L., and Chris, C. Configuring Huge Pages in Red Hat Enterprise Linux 4 or 5. https://goo.gl/lqB1uf, 2014.

[12] Bovet, D. P., and Cesati, M. *Understanding the Linux kernel.* O'Reilly Media, Inc., 2005.

[13] Chang, K., et al. Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM. In *HPCA* (2016).

[14] Chang, K., et al. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *SIGMETRICS* (2016).

[15] Chang, K., et al. Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms. *SIGMETRICS* (2017).

[16] Chatterjee, N., et al. Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access. In *MICRO* (2012).

[17] Chi, P., et al. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *ISCA* (2016).

[18] Chou, C., et al. CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache. In *MICRO* (2014).

[19] Chou, C., et al. BATMAN: Maximizing Bandwidth Utilization of Hybrid Memory Systems. Tech report, ECE, Georgia Institute of Technology, 2015.

[20] Chou, C., et al. BEAR: Techniques for Mitigating Bandwidth Bloat in Gigascale DRAM Caches. In *ISCA* (2015).

[21] Chou, C., et al. CANDY: Enabling Coherent DRAM Caches for Multi-Node Systems. In *MICRO* (2016).

[22] Dhiman, G., et al. PDRAM: a Hybrid PRAM and DRAM Main Memory System. In *DAC* (2009).

[23] Franey, S., and Lipasti, M. Tag Tables. In *HPCA* (2015).

[24] Gulur, N., et al. Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth. In *MICRO* (2014).

[25] Henning, J. L. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News 34*, 4 (2006).

[26] Huang, C.-C., et al. C^3D: Mitigating the NUMA Bottleneck via Coherent DRAM Caches. In *MICRO* (2016).

[27] Huang, C.-C., and Nagarajan, V. ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache. In *PACT* (2014).

[28] Jang, H., et al. Efficient Footprint Caching for Tagless DRAM Caches. In *HPCA* (2016).

[29] JEDEC. JESD235 High Bandwidth Memory (HBM) DRAM, 2013.

[30] Jeffers, J., et al. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[31] Jevdjic, D., et al. Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *ISCA* (2013).

[32] Jevdjic, D., et al. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *MICRO* (2014).

[33] Jiang, X., et al. CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms. In *HPCA* (2010).

[34] Kim, Y., et al. Ramulator: A Fast and Extensible DRAM Simulator. *CAL* (2016).

[35] Kumar, S., and Wilkerson, C. Exploiting Spatial Locality in Data Caches using Spatial Footprints. In *ISCA* (1998).

[36] Lee, D., et al. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE transactions on Computers* (2001).

[37] Lee, D., et al. Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *HPCA* (2013).

[38] Lee, Y., et al. A Fully Associative, Tagless DRAM Cache. In *ISCA* (2015).

[39] Li, Y., et al. Utility-Based Hybrid Memory Management. In *CLUSTER* (2017).

[40] Liptay, J. Structural Aspects of the System/360 Model 85, II: The cache. *IBM Systems Journal* (1968).

[41] Loh, G. H., and Hill, M. D. Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches. In *MICRO* (2011).

[42] Lu, S.-L., et al. Improving DRAM Latency with Dynamic Asymmetric Subarray. In *MICRO* (2015).

[43] Luo, Y., et al. Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory. In *DSN* (2014).

[44] Meswani, M., et al. Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-stacked and Off-package Memories. In *HPCA* (2015).

[45] Meza, J., et al. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. *CAL* (2012).

[46] O'Connor, M. Highlights of the High-Bandwidth Memory (HBM) Standard.

[47] Phadke, S., and Narayanasamy, S. MLP Aware Heterogeneous Memory System. In *DATE* (2011).

[48] Qureshi, M., et al. A Case for MLP-Aware Cache Replacement. *ISCA* (2006).

[49] Qureshi, M., et al. Adaptive Insertion Policies for High Performance Caching. In *ISCA* (2007).

[50] Qureshi, M., and Loh, G. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical DRAM-Tags with a Simple and Practical Design. In *MICRO* (2012).

[51] Robinson, J., and Devarakonda, M. Data cache management using frequency-based replacement. In *SIGMETRICS* (1990).

[52] Rothman, J., and Smith, A. Sector Cache Design and Performance. In *MASCOTS* (2000).

[53] Sanchez, D., and Kozyrakis, C. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *ISCA* (2013).

[54] Sim, J., et al. Transparent Hardware Management of Stacked DRAM as Part of Memory. In *MICRO* (2014).

[55] Sodani, A. Intel®Xeon Phi™ Processor "Knights Landing" Architectural Overview. https://goo.gl/dp1dVm, 2015.

[56] Sodani, A., et al. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro 36*, 2 (2016), 34–46.

[57] Stallings, W., et al. *Operating Systems: Internals and Design Principles*, vol. 148. Prentice Hall Upper Saddle River, NJ, 1998.

[58] Villavieja, C., et al. DiDi: Mitigating The Performance Impact of TLB. Shootdowns Using A Shared TLB Directory. In *PACT* (2011).

[59] Yoon, H., et al. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *ICCD* (2012).

[60] Yu, X., et al. IMP: Indirect Memory Prefetcher. In *MICRO* (2015).

[61] Yu, X., et al. Banshee: Bandwidth-Efficient DRAM Caching Via Software/Hardware Cooperation. *arXiv:1704.02677* (2017).